Adaptación de la herramienta PMCTrack a entornos de producción

Adaptation of the PMCTrack tool to production systems

TRABAJO FIN DE GRADO

Lázaro Clemen Palafox

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid
2021

Adaptación de la herramienta PMCTrack a entornos de producción

Memoria de Trabajo Fin de Grado

Lázaro Clemen Palafox

Dirigido por: Juan Carlos Sáez Alcaide

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid
2021

Resumen

PMCTrack es una herramienta de código abierto para Linux que permite la monitorización de diversos aspectos de las aplicaciones en tiempo de ejecución empleando contadores *hardware* u otros recursos *hardware* de monitorización, como sensores de temperatura o de consumo energético. Para ofrecer acceso a la información de monitorización al usuario final y al sistema operativo (SO), esta herramienta consta de componentes tanto de espacio de usuario como de kernel. Un aspecto diferencial de PMCTrack con respecto a otras herramientas es su sencilla interfaz de acceso a los datos de monitorización desde el kernel Linux. Esta interfaz, que es independiente de arquitectura, facilita la realización de optimizaciones en el SO, principalmente orientadas a la planificación de procesos y a la gestión de recursos.

A pesar de las numerosas ventajas que ofrece la versión más reciente de PMCTrack, el hecho de requerir un parche en el kernel Linux para funcionar, así como su integración aún parcial con el subsistema perf_events de Linux para algunos procesadores de Intel, hace que en la práctica su adopción resulte complicada en sistemas de producción, donde la inclusión de parches del núcleo no es siempre viable. El objetivo de este Trabajo Fin de Grado ha sido subsanar estas dos limitaciones. En particular, para la eliminación del parche del kernel, se ha realizado un estudio exhaustivo de los sistemas de trazado (*tracing*) de Linux y –haciendo uso de estos sistemas–, se ha procedido a incorporar la funcionalidad del parche de PMCTrack de forma muy eficiente desde un módulo cargable del kernel. Esto posibilita la completa eliminación de dicho parche. Asimismo, se ha ampliado sustancialmente la funcionalidad del backend de *perf* de PMCTrack (capa de integración con perf_events) para lograr el acceso a un mayor número de eventos de monitorización en muy diversas arquitecturas. Mediante un estudio experimental empleando benchmarks de SPEC CPU2006 y CPU2007, se demuestra que las nuevas características incorporadas en PMCTrack, que formarán parte de su nueva versión 2.0, no afectan al rendimiento ni la precisión de esta herramienta, y al mismo tiempo eliminan por completo las limitaciones anteriormente citadas.

palabras clave: PMCTrack, contadores hardware, kernel Linux, ftrace, tracepoints, perf_events, livepatching, sistemas de trazado.

Abstract

PMCTrack is an open source tool for Linux that allows the monitoring of various aspects of applications at runtime using hardware counters or other hardware monitoring resources, such as temperature or power consumption sensors. To provide access to monitoring information to the end user and the operating system (OS), this tool consists of both user space and kernel components. A distinguishing feature of PMCTrack compared to other tools is its simple interface for accessing monitoring data from the Linux kernel. This interface, which is architecture-independent, makes it easy to perform OS optimizations, mainly oriented to process scheduling and resource management.

Despite the many advantages offered by the latest version of PMCTrack, the fact that it requires a Linux kernel patch to work, as well as its still partial integration with the Linux perf_events subsystem for some Intel processors, makes its adoption complicated in practice on production systems, where the inclusion of kernel patches is not always feasible. The objective of this Final Degree Project has been to overcome these two limitations. In particular, for the removal of the kernel patch, an exhaustive study of Linux tracing systems has been carried out and by using these systems the functionality of the PMCTrack patch has been incorporated, in a very efficient way, into a loadable kernel module. This makes it possible to completely remove the patch. In addition, the functionality of PMCTrack's *perf* backend (perf_events integration layer) has been substantially extended to provide access to a larger number of monitoring events in a wide variety of architectures. Through an experimental study using SPEC CPU2006 and CPU2007 benchmarks, it is proven that the new features incorporated in PMCTrack, which will be part of its new version 2.0, do not affect the performance and accuracy of this tool, and at the same time completely eliminate the aforementioned limitations.

palabras clave: PMCTrack, hardware counters, Linux kernel, ftrace, tracepoints, perf_events, livepatching, tracing systems.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

Esta introducción presenta la motivación que ha llevado al planteamiento de este trabajo de fin de grado, expone los objetivos a alcanzar en el proyecto y su planificación. También se describe la estructura de esta memoria.

1.1 Motivación

Los contadores *hardware* de monitorización del rendimiento o PMCs (*Performance Monitoring Counters*), son un conjunto de registros del procesador que permiten la captura de eventos para la medición de métricas de rendimiento relevantes, como las instrucciones por ciclo, la tasa de fallos de distintos niveles de caché o los ciclos de parada del *pipeline* del procesador causados por distintos motivos. Un aspecto notable de los contadores *hardware* es el hecho de que en los procesadores actuales, éstos sólo son directamente accesibles desde el nivel de privilegio donde típicamente se ejecuta el sistema operativo (SO). Esto hace que, para acceder a la información de monitorización desde espacio de usuario, sea necesario disponer de soporte específico en el sistema operativo, implementado como parte del kernel o en un driver dedicado.

Entre las distintas herramientas que permiten acceder a los PMCs se encuentra PMCTrack [1]. Se trata de una herramienta de código abierto para GNU/Linux cuyo desarrollo comenzó en el año 2007. PMCTrack es la única herramienta de monitorización diseñada desde su origen para ofrecer datos de los PMCs, así como de otras fuentes de monitorización *hardware* de un sistema, a los componentes del SO. Esta información de monitorización se expone de forma sencilla mediante una API del núcleo que emplea abstracciones independientes de la arquitectura. Asimismo, al igual que otras herramientas de gestión de contadores *hardware*, como `perf` o Intel Performance Counter Monitor [2], [3], PMCTrack también permite la recopilación de datos de monitorización desde el espacio de usuario. Para ello proporciona el programa `pmctrack` –una herramienta de línea de comandos–, PMCTrack-GUI –un *frontend* gráfico– y *libpmctrack* –una biblioteca

que permite la instrumentación de código C/C++–. Cabe también destacar que cualquier tipo de información de monitorización relevante proporcionada por el *hardware*, pero no expuesta directamente a través los PMCs (como el consumo energético, la ocupación de cache, o el número de fallos de página, etc.), también puede exponerse al SO y al usuario final a través de la abstracción de *contadores virtuales* de PMCTrack [1].

Este Trabajo Fin de Grado (TFG) se centra en la extensión de la funcionalidad de PMCTrack que se implementa a nivel del kernel. Para la correcta monitorización del rendimiento de cada aplicación de forma individual usando PMCs, el kernel del SO ha de gestionar los registros asociados a estos recursos de monitorización *hardware*. Más concretamente, cuando un hilo de ejecución cuyo rendimiento está siendo monitorizado sale de la CPU (p.ej., se bloquea por entrada-salida o sincronización) es preciso salvar el contexto de los contadores *hardware*. Análogamente, este contexto ha de restaurarse en la CPU adecuada cuando el SO dé la oportunidad de nuevo al hilo de ejecutar instrucciones. Esto pone de manifiesto que el código de gestión de los contadores *hardware* está íntimamente ligado al del componente del kernel Linux que efectúa los cambios de contexto: el planificador de procesos.

Teniendo esto en cuenta, para realizar el mantenimiento de los PMCs y de otros registros de monitorización *hardware* de forma directa, toda herramienta de monitorización ha de ser plenamente consciente de los eventos críticos de planificación (como la creación de un proceso, su terminación, la ocurrencia de cambios de contexto, etc.). Esto implica, que para el mantenimiento de los PMCs en PMCTrack, es preciso llevar a cabo cambios específicos en el planificador de Linux, que posibiliten la "captura" de estos eventos del planificador. Aunque introducir cambios en el kernel es la forma más natural de lograr acceso directo a los contadores *hardware*, también supone una importante limitación: toda modificación futura en la herramienta de gestión de contadores exige la recompilación del kernel, y el reinicio del sistema para su puesta a prueba. Esto hace que el proceso de desarrollo pueda llegar a ser extremadamente lento.

A diferencia de una herramienta de gestión de contadores *hardware* que se implementa directamente en el kernel Linux, como es el caso de la herramienta `perf` introducida en el Linux *mainline* –con su consecuente elevado coste de mantenimiento–, PMCTrack divide su funcionalidad a nivel de kernel en dos partes:

- **Una API (*Application Programming Interface*) en el kernel.** Esta API consta de (1) una serie de llamadas en el planificador para la captura de eventos de planificación que conforma una interfaz de callbacks, (2) una API específica para asociar *callbacks* a estos eventos desde otros componentes del kernel, (3) funciones para acceder a los datos de los PMCs por hilo desde dentro del kernel y (4) la inclusión de campos extra en el descriptor de proceso (`struct task_struct`) para el mantenimiento de la información de monitorización. Esta API de PMCTrack se introduce en el kernel a través de un parche. Por lo tanto, para usar PMCTrack en un sistema es necesario disponer de un kernel modificado con este parche.

Cabe destacar que el parche del kernel de PMCTrack no supone la inclusión de cambios muy significativos en las fuentes de Linux; tan solo se han de incorporar dos nuevos ficheros fuente y añadir unas cuantas decenas de líneas de código en las fuentes del planificador de Linux y ficheros relacionados [1].

- **Un módulo del kernel que implementa la mayor parte de la funcionalidad de PMCTrack.** Este módulo del kernel, al cargarse, instala una interfaz de callbacks para recibir las notificaciones necesarias del planificador que posibilitan la gestión de los registros de monitorización *hardware*. Cabe destacar que el acceso a los contadores *hardware* de rendimiento en distintas arquitecturas y modelos de procesador en PMCTrack, se abstrae a otras capas de *software* de PMCTrack mediante el uso de *backends* –uno por cada arquitectura–. En tiempo de compilación se generan distintas variantes del módulo del kernel para cada arquitectura soportada [4].

El hecho de que la mayor parte de la funcionalidad de PMCTrack esté incluida en un módulo del kernel simplifica enormemente el mantenimiento y depuración de esta herramienta. Cualquier nueva funcionalidad puede incluirse en el código del módulo del kernel, y ponerse a prueba de forma rápida cargando y descargando el módulo en Linux las veces que sean necesarias, todo ello sin tener que reiniciar el sistema[1]. A pesar de su facilidad de mantenimiento, PMCTrack cuenta con dos limitaciones relevantes, que dificultan su adopción generalizada en entornos de producción:

1. **Dependencia de un parche del kernel Linux para funcionar.**

   Esto conlleva la creación y publicación, por parte de los desarrolladores de PMCTrack, de un parche del kernel Linux específico para cada versión de Linux y la arquitectura sobre la que se desee trabajar. Un usuario con intención de usar PMCTrack sobre una arquitectura y versión de Linux para los cuales ya exista un parche oficial de PMCTrack, deberá tener conocimientos básicos en compilación del kernel Linux para lograr la correcta instalación del parche y la compilación del kernel.

   En el caso de que no exista un parche de PMCTrack publicado por los desarrolladores del mismo, un usuario que quiera hacer uso de PMCTrack deberá disponer, además de conocimientos sobre la compilación del kernel, de un conocimiento avanzado sobre PMCTrack y el kernel Linux. Al fin y al cabo, para la correcta implementación de un parche –en base a un parche existente para una versión anterior de Linux– es preciso hacer un seguimiento de los cambios sufridos por el código del planificador en la nueva versión para determinar dónde incluir las nuevas líneas de código del parche (principalmente notificaciones al módulo del kernel de la ocurrencia de eventos del planificación).

[1] Sólo errores críticos en la implementación que causan un error del núcleo (*kernel panic*) obligan a realizar dichos reinicios.

2. **Integración parcial con el subsistema perf_events de Linux.**

   Las herramientas de gestión de contadores *hardware* creadas con anterioridad a la década de 2010, como es el caso de PMCTrack, se caracterizan por realizar un acceso directo a los registros de monitorización *hardware* [5]. Como se discutirá en detalle en el capítulo 2, esto ha provocado históricamente una incompatibilidad entre distintas herramientas de gestión de contadores en Linux. Para tratar de resolver este problema se creó el subsistema perf_events de Linux [6], que ofrece una API de bajo nivel para realizar el acceso y la gestión de los contadores abstrayendo la mayor parte de aspectos específicos de arquitectura.

   Un primer paso para la integración entre PMCTrack y perf_events, fue la creación de un backend genérico (*mchw_perf*) en el que el acceso a los PMCs no se realiza directamente mediante la lectura de los registros de la PMU (**Performance Monitoring Unit**) –como en el resto de *backends* de PMCTrack–, sino empleando la API de perf_events [7]. No obstante, este nuevo backend, incorporado en la última versión estable de PMCTrack (v1.5) no está completo y carece de muchas funcionalidades básicas para PMCTrack. En particular, tiene sólo soporte para algunos procesadores x86 de Intel, implementa únicamente la funcionalidad de PMCTrack para el espacio de usuario y es compatible sólo con un conjunto muy reducido de los eventos ofrecidos por el subsistema perf_events de Linux.

Estas dos limitaciones hacen que en la práctica sea complicada la adopción generalizada de PMCTrack en entornos de producción, y que la creación de soporte para nuevas arquitecturas y modelos de procesador se pueda convertir en un proceso lento y costoso. En entornos de producción, disponer de kernels estables con actualizaciones de seguridad es esencial para ofrecer la robustez necesaria que demandan este tipo de entornos. Requerir de un kernel modificado, en lugar de uno estable (*longterm*) complica enormemente la aplicación de actualizaciones de seguridad, lo que trae consigo reticencias justificadas de los administradores de sistemas. Por otra parte cabe destacar que para incorporar soporte para nuevas arquitecturas y modelos de procesador en PMCTrack, tradicionalmente era preciso crear un nuevo backend o modificar uno existente, para realizar el acceso a bajo nivel a los contadores *hardware* empleando código ensamblador (incrustado en código "C"). La creación del nuevo –aunque limitado– backend de perf_events en PMCTrack v1.5 supone una gran oportunidad para ofrecer soporte de PMCTrack más rápidamente a nuevas arquitecturas; por el hecho de que perf_events es mantenido por empleados de los principales fabricantes de *hardware* para dar soporte a un amplio espectro de arquitecturas y modelos de procesador.

PMCTrack es la única herramienta en la actualidad que ofrece un acceso sencillo e independiente de la arquitectura a los datos de monitorización *hardware* dentro de componentes del kernel Linux. Esto supone que facilitar su adopción a entornos de producción, así como simplificar el proceso de inclusión de soporte en nuevas arquitecturas y modelos de procesador suponga una enorme motivación. Aunque esto puede suponer un gran reto –y la eliminación de limitaciones históricas de PMCTrack–, hay dos factores

que hacen viable abordarlo en la actualidad. En primer lugar, el kernel Linux implementa actualmente un conjunto de tecnologías de trazado –como *tracepoints* [8], *Kprobes* [9] y *ftrace* [10]– que han alcanzado un alto grado de madurez y gozan de soporte en un buen número de arquitecturas. Este tipo de tecnologías permiten, entre otras cosas, conocer qué sucede dentro de Linux (qué funciones se ejecutan y en qué orden) o incluso modificar el comportamiento del kernel en tiempo de ejecución (p.ej. *livepatching* [11]) mediante la instalación de *callbacks* en determinados puntos del código del kernel. El uso de estas tecnologías de trazado plantea la posibilidad de eliminar la limitación de tener que aplicar un parche de PMCTrack al kernel. Por otra parte, el backend de perf_events en PMCTrack v1.5, a pesar de sus limitaciones, supuso una excelente prueba de concepto de que el uso de este subsistema del kernel para evitar el acceso a bajo nivel a los PMCs era posible [7]. Por lo tanto, la extensión de la funcionalidad de este backend constituye un gran incentivo para acelerar el proceso de incorporar soporte a otras arquitecturas y modelos de procesador –en particular ARM y AMD–, así como para permitir el acceso desde PMCTrack a un mayor espectro de los eventos de monitorización ofrecidos por perf_events.

1.2 Objetivos

El objetivo principal de este proyecto es lograr adaptar la herramienta PMCTrack a entornos de producción. Lograr esto implica perseguir los tres siguientes subobjetivos:

1. La adaptación de la instalación de la API de PMCTrack en el kernel Linux a una implementación que no requiera modificar el código fuente del kernel. Esto exige el estudio exhaustivo, la elección y la aplicación combinada de las distintas tecnologías de trazado existentes en la rama principal de Linux, así como el análisis de las distintas alternativas para lograr la mayor eficiencia en la implementación.
2. Ampliar las arquitecturas soportadas por el backend experimental de perf_events incluido en la versión v1.5 de PMCTrack. Ésto supone la extensión de este backend a nuevas arquitecturas, como son ARMv7 y ARMv8, así como la inclusión de soporte para nuevos procesadores de AMD.
3. Expandir la funcionalidad del backend experimental de perf_events. Ésto implica un aumento sustancial del número de eventos del subsistema de perf_events que podrían monitorizarse con PMCTrack, tanto desde espacio de usuario, como desde componentes del kernel.

1.3 Plan de trabajo

En la planificación de este proyecto se han establecido una serie de tareas para lograr los objetivos planteados. El tiempo que ha durado la realización de cada tarea se representa en el diagrama de Gantt de la figura 1.1. A continuación se describe cada tarea del diagrama, que se identifica mediante un número en el siguiente listado:

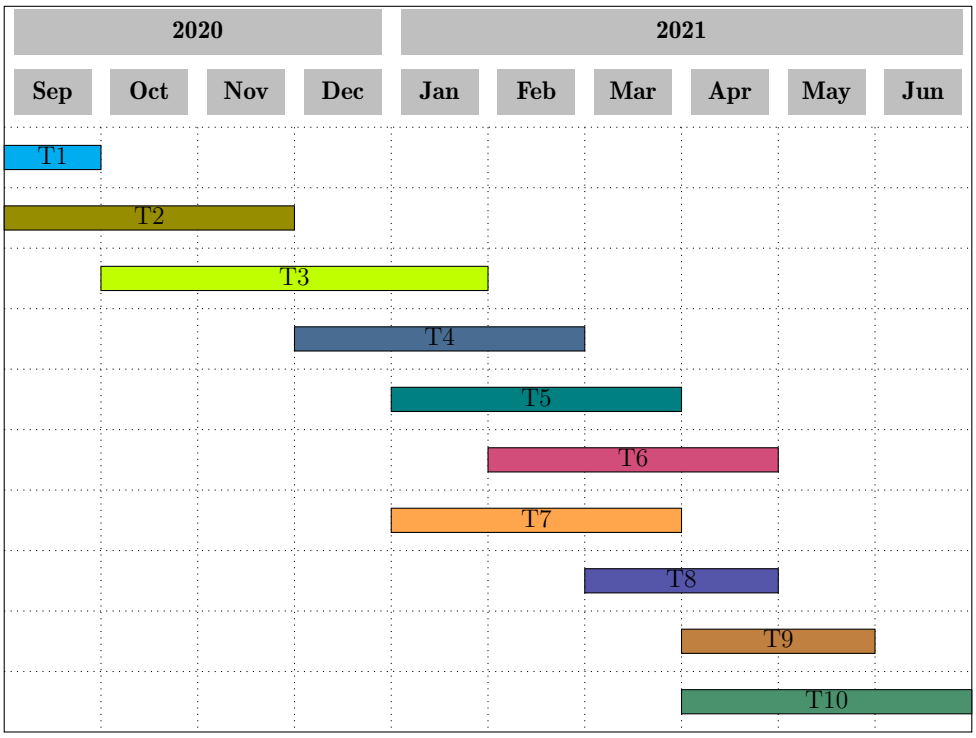


Figura 1.1: Tiempo invertido para cada tarea realizada durante el proyecto

T1. Planteamiento del proyecto. Esta tarea corresponde a la fase previa al desarrollo. En ella se concretaron las limitaciones de PMCTrack y su necesidad de ser eliminadas para poder convertir PMCTrack en una versión apta para entornos de producción. Se estudió cómo dividir en partes el parche del kernel de PMCTrack para poder abordar en orden las partes del parche que plantean adaptaciones distintas. Se hizo una puesta en común de posibles tecnologías a usar en el proyecto y se planificó el orden en el que se tratarían las partes en las que se dividió el parche de PMCTrack.

T2. Estudio de tecnologías de trazado. En esta tarea se hizo el estudio de los sistemas de trazado (tracepoints [8], Kprobes [9] y ftrace [10]). Se determinó si ofrecen la funcionalidad y eficiencia necesaria para abordar la adaptación de la API de PMCTrack del kernel a una implementación que no requiera de un parche del kernel. Esta fase incluye la creación de módulos del kernel para poner a prueba el comportamiento de dichos sistemas de trazado, de forma que se pudiese evaluar qué tecnologías son las más adecuadas.

T3. Implementación de la adaptación de la interfaz de callbacks de la API de PMCTrack. Esta tarea conllevó la aplicación de dichos sistemas de trazado escogidos para adaptar la interfaz de *callbacks* de la API de PMCTrack a una implementación sin necesidad de un parche para Linux. La implementación se hizo con la idea de que funcionara tanto con un Linux *vanilla* como con un kernel modificado con

la API de PMCTrack. Manteniendo así compatibilidad hacia atrás y con otros backends de PMCTrack.

T4. Estudio del subsistema perf_events de Linux. En esta etapa se estudió la posibilidad que pueda ofrecer el subsistema perf_events para conseguir mantener los datos de monitorización necesarios para PMCTrack en la información asociada a cada tarea de Linux.

T5. Almacenamiento de datos de monitorización por hilo en kernels *vanilla*. En esta tarea se realizó la implementación, mediante el uso de perf_events, de la nueva forma en la que PMCTrack almacena los datos de monitorización por tarea, de manera que no requiera de un parche del kernel Linux. De la misma forma que con la interfaz de callbacks de PMCTrack, esta implementación se realizó también con la idea de mantener compatibilidad hacia atrás.

T6. Terminación de compatibilidad en la capa independiente de arquitectura. Esta tarea consistió en la realización de los últimos pasos de la adaptación a la nueva implementación de la API de PMCTrack, para que su capa independiente de arquitectura siguiera funcionando correctamente con independencia del kernel empleado. Se debía mantener la compatibilidad tanto si PMCTrack se ejecuta sobre la nueva implementación de la API de PMCTrack o por el contrario sobre kernels parcheados con la antigua implementación.

T7. Extensión de la funcionalidad del backend de perf_events. En esta etapa se adaptó la nueva implementación de la API de PMCTrack y el backend experimental de perf_events para crear soporte para procesadores ARM y AMD. Esto también supuso la extensión del backend experimental de perf_events para lograr un soporte mucho más amplio de los eventos que dispone perf_events.

T8. Compatibilidad de la nueva implementación con los backends *legacy*. En esta tarea se implementaron las modificaciones necesarias para lograr la compatibilidad con antiguos backends de PMCTrack (`intel-core`, `amd`, `arm`, `odroid-xu`, etc.). Se consiguió así el correcto funcionamiento de todos los backends de PMCTrack publicados con la nueva implementación de la API de PMCTrack.

T9. Evaluación experimental y análisis. Se hizo un estudio de la posible sobrecarga que incorpora el uso de sistemas de trazado para la nueva implementación de la API de PMCTrack. Se hizo el análisis experimental de esta sobrecarga empleando benchmarks de las suites SPEC CPU2006 y CPU2017.

T10. Elaboración de la memoria. En esta fase se realizó la redacción y revisión continua de esta memoria.

1.4 Organización de esta memoria

A continuación, se plantea una breve descripción de los distintos capítulos que conforman el resto de esta memoria:

- En el **capítulo 2** se describe el contexto histórico y la arquitectura de PMCTrack. También se explica en detalle el parche de PMCTrack, la API de PMCTrack del kernel, y su división en partes para abordar el proyecto.
- En el **capítulo 3** se expone el contexto histórico de los sistemas de trazado en Linux y se describe los tipos de trazado que existen dentro del kernel Linux. Además se expone la existencia de sistemas de trazado dentro de la rama principal del kernel Linux que permiten hacer trazado sobre el código fuente de Linux en ejecución. También se incluye la explicación del funcionamiento interno de dichos sistemas de trazado y la argumentación sobre las funcionalidades que ofrecen en relación a su aplicación en la adaptación de la API de PMCTrack.
- En el **capítulo 4** se explica en detalle la implementación de la nueva versión de la API de PMCTrack compatible con entornos de producción. La implementación se realiza mediante el uso de los sistemas de trazado estudiados, ftrace, tracepoints y perf_events.
- El **capítulo 5** proporciona la explicación detallada de los cambios realizados sobre el backend de perf_events experimental para su extensión de funcionalidad y soporte para procesadores más allá de Intel x86. También se presentan los cambios realizados sobre la nueva implementación de la API de PMCTrack para ofrecer compatibilidad con otras arquitecturas (AMD y ARM) y para el resto de backends con los que cuenta PMCTrack.
- En el **capítulo 6** se discuten los resultados experimentales obtenidos empleando benchmarks de las suites SPEC CPU2006 y CPU2007. Se realiza también un análisis detallado de la precisión de las métricas de rendimiento obtenidas y de la sobrecarga asociada a la nueva implementación de la interfaz de callbacks de la API de PMCTrack.
- En el **capítulo 7** se ponen de manifiesto las ventajas de la nueva versión de PMCTrack. También se exponen las conclusiones y conocimientos adquiridos durante el desarrollo de este trabajo de fin de grado. Se incluyen distintas ideas para la posible continuación del proyecto realizado.
- En los **apéndices A y B** puede encontrarse la traducción a inglés de está introducción (capítulo 1) así como de las conclusiones (capítulo 7), respectivamente.

Capítulo 2

PMCTrack

Este capítulo entra en detalle en el contexto histórico en el surgió PMCTrack, la arquitectura interna de PMCTrack y la descripción del parche del kernel de PMCTrack.

El capítulo se estructura en las siguientes secciones. En la **sección 2.1** se hace un recorrido en la historia de las herramientas de monitorización del rendimiento en Linux en relación a la creación de PMCTrack. En la **sección 2.2** se expone en detalle la arquitectura interna de la que se compone PMCTrack. Por último en la **sección 2.3** se hace una división en función de las características de la diferentes funcionalidades que comprenden el parche de PMCTrack. También se explica de forma detallada las distintas partes en las que se divide el parche de PMCTrack, presentando posibles aproximaciones a sus adaptaciones para que no requieran de un parche del kernel Linux.

Todas las rutas a las que se hace referencia a lo largo de este documento toman como directorio raíz las fuentes de Linux *mainline*, salvo que se indique lo contrario.

2.1 Contexto histórico

Para entender el contexto histórico en que surgió PMCTrack y argumentar así la necesidad de su existencia con respecto al resto de herramientas de monitorización, es necesario retroceder en el tiempo y analizar los inicios de la aparición de herramientas de monitorización del rendimiento.

La mayoría de las CPUs modernas incluyen contadores *hardware* de rendimiento, PMCs. Poco después del lanzamiento del procesador Pentium original [12] aparecieron parches que proporcionan soporte en Linux para los contadores de rendimiento [5]. Para aquel entonces los fabricantes de procesadores apenas proporcionaban documentación detallada sobre el uso de los PMCs, lo que llevó a que apenas se crearan herramientas de monitorización. Dado que sólo se puede acceder a los PMCs directamente a nivel de privilegio del sistema operativo, es necesario desarrollar herramientas a nivel del kernel

para que el usuario final y los programas del espacio de usuario puedan acceder a los PMCs [1].

Desde la introducción del soporte de PMCs hasta mediados de la década de los 2000 empezaron a surgir herramientas de monitorización como: OProfile 2002 [13], Perfmon2 2003 [14] o incluso anteriormente en 1999 Perfctr [5]. Aunque estas herramientas de monitorización de rendimiento simplificaban en gran medida la recopilación de datos de PMCs para aplicaciones desde el espacio de usuario, no proporcionaban un mecanismo a nivel de kernel, independiente de la arquitectura, que fuera capaz de exponer las métricas de los PMCs a los componentes del sistema operativo [1].

Fue entonces en el año 2007, en el contexto de la tesis doctoral de Juan Carlos Sáez Alcaide (en el Departamento de Arquitectura de Computadores y Automática (DACyA) de la Universidad Complutense de Madrid (UCM)), cuando se creo la primera versión de PMCTrack. En el inicio PMCTrack era una herramienta propietaria para Linux que proporcionaba un mecanismo sencillo e independiente de la arquitectura para hacer posible que el planificador del sistema operativo accediera a los datos de los PMCs por hilo [1], [15]. Todo ello estaba inspirado por la necesidad de tener una herramienta capaz de proporcionar al planificador del kernel Linux acceso a los contadores *hardware* para realizar optimizaciones en tiempo de ejecución.

A lo largo de los siguientes años continuaron apareciendo herramientas de monitorización del rendimiento pero al igual que las ya existentes requerían sus propios parches para el kernel para su funcionamiento [16], [17]. La mayoría de ellas se centraban sólo en la arquitectura x86 y todas ellas, al contrario que la herramienta desarrollada en el DACyA, centradas en una monitorización externa para exponer al usuario los datos de los PMCs. También se produjo la inclusión, dentro de las fuentes del kernel Linux, de la herramienta OProfile pero ésta no fue utilizada por toda la comunidad de Linux y por tanto se mantuvo el desarrollo del resto de herramientas de monitorización del rendimiento con parches para el kernel.

A finales del año 2008 Ingo Molnar y Thomas Gleixner anunciaron la inclusión, en Linux 2.6.31, de una nueva implementación de soporte de contadores de rendimiento para Linux [18]. Ésta recibió el nombre de *Performance Counters for Linux* (PLC) e introdujo también su propia herramienta de línea de comandos `perf` [19], [20]. Posteriormente PLC fue renombrado como perf_events en la versión 2.6.32 en 2009. El subsistema perf_events se creó en respuesta a una propuesta de integración de perfmon2 en el kernel [5], proponiendo un framework común dentro del kernel que proporcionase abstracción de las capacidades *hardware* de los contadores de rendimiento [5], [18]. Con ello se introdujo también el subsistema de eventos perf_events que incluía una API del kernel basada en una abstracción única de contador, proporcionando contadores por tarea y por CPU [18]. Esta API en un principio causó cierto rechazo y crítica dentro de la comunidad Linux [21], [22]. Su implementación estaba lejos de ser completa y fallaba en la compatibilidad con CPUs que no fuesen Intel o antiguas CPUs Intel, pero el enfoque básico ya estaba ahí y era estable [18]. Este enfoque se centraba en ofrecer una API del

kernel diseñada principalmente para construir sobre ella herramientas de monitorización orientadas al espacio del usuario.

En el año 2012 se realiza la primera publicación sobre PMCTrack [23]. Ésta vino de la mano del trabajo de fin de carrera de unos estudiantes de la Universidad Complutense de Madrid. Éstos crearon la primera versión de PMCTrack con soporte de monitorización en el espacio de usuario, junto con la ya presente en el espacio del kernel [23].

Para entonces el subsistema eventos perf_events tenía una gran actividad de desarrollo. Aunque la introducción de `perf` fue difícil de aceptar para los usuarios de herramientas alternativas de monitorización del rendimiento, la mayoría ya habían aceptado `perf` [24]. A pesar del potencial de `perf` y de las otras herramientas mencionadas, ninguna de ellas implementaban un mecanismo a nivel de kernel que estuviese específicamente adaptado para crear implementaciones, independientes de la arquitectura, de estrategias de planificación que aprovechen los datos de los PMCs para sus decisiones internas, como el caso de PMCTrack [1].

No fue hasta Septiembre de 2015 que se saca a la luz la primera versión pública de PMCTrack, liberándose su código fuente [25]. A partir de entonces se han ido introduciendo mejoras a PMCTrack, incluyendo nuevas funcionalidades como la conexión de PMCTrack a un proceso en ejecución (modo *attach*), nuevo soporte para arquitecturas y soporte para nuevas versiones del kernel Linux [4].

Avanzando al año 2019, perf_events no sólo pasó a ser compatible con una amplia gama de arquitecturas de procesadores, sino que también ofrecía ya a los usuarios sorprendentes capacidades de trazado (*tracing*) de *software*, –a mano del uso de sistemas de trazado como ftrace, tracepoints y kprobes– permitiendo, entre otras funcionalidades, realizar un seguimiento de las llamadas al sistema de un proceso o de la actividad relacionada con el planificador [1]. La API de perf_events, a pesar de su difícil uso interno desde el dentro el propio kernel Linux, ofrecía ya la suficiente versatilidad como para plantear su integración en PMCTrack. De ahí surgió la última versión estable de PMCTrack, v1.5, que, entre otras mejoras, añadió el backend perf experimental para PMCTrack [4]. Este backend presentó una implementación del acceso a los contadores *hardware* de rendimiento a través de la API del kernel de perf_events, con el intento de eliminar la dependencia arquitectónica de la que sufren el resto de backends de PMCTrack. La versión v1.5 de PMCTrack, publicada en 2020, es el punto de partida para este trabajo de fin de grado.

Con este nuevo *backend* perf de PMCTrack se logró probar el concepto de que PMCTrack puede hacer uso del subsistema de eventos perf_events para el acceso a los contadores *hardware*, eliminando la necesidad de hacer el acceso directamente con backends adaptados a cada arquitectura. El backend experimental de perf_events sólo ofrece soporte para arquitecturas x86 en procesadores Intel y sólo implementa la funcionalidad de PMCTrack para el espacio de usuario. También presenta la falta de compatibilidad de modos de funcionamiento de PMCTrack como son el modo *attach* o la monitorización

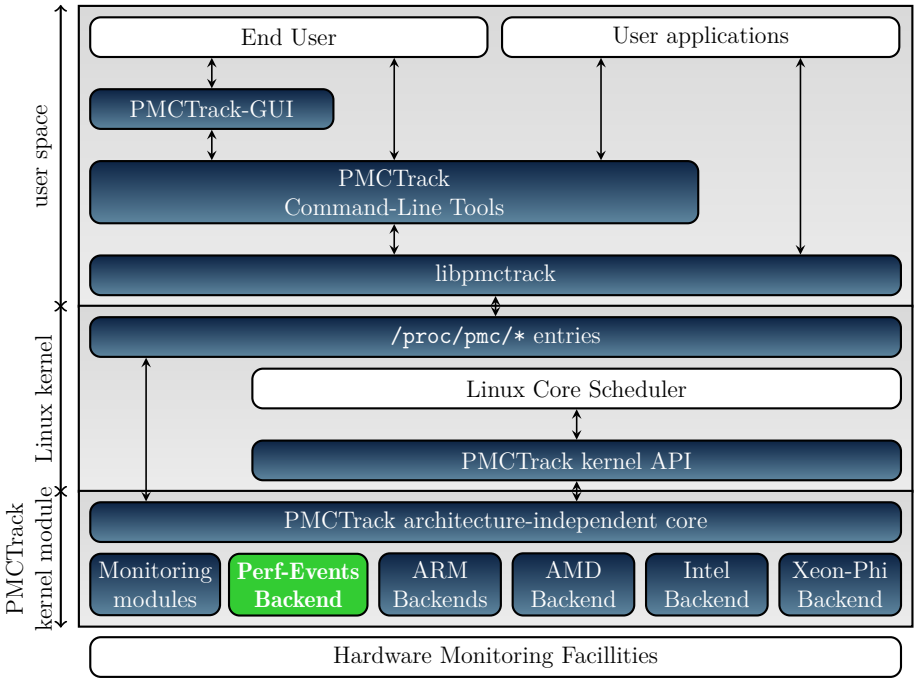


Figura 2.1: Arquitectura interna de PMCTrack v1.5

de aplicaciones lanzadas a ejecución en procesadores con *cores* asimétricos.

Llegada la versión v1.5 de PMCTrack, ésta ya había sufrido múltiples fases en su desarrollo que hicieron que PMCTrack ya no se centrara sólo en ofrecer datos de monitorización a planificadores dentro del kernel, sino que esta funcionalidad se había generalizado para ofrecer datos de monitorización de manera sencilla e independiente de la arquitectura a cualquier componente de Linux que quiera hacer uso de ellos.

Conseguido demostrar el posible uso de perf_events junto con PMCTrack con el nuevo backend perf –presentando la posibilidad de eliminar la dependencia arquitectónica de los backends de PMCTrack– ya se estaba un paso más adelante de la posibilidad de adaptar PMCTrack a entornos de producción. La última barrera para conseguir la adaptación completa de PMCTrack a entornos de producción es el parche de PMCTrack para la instalación de la API del kernel de PMCTrack dentro del kernel Linux.

2.2 Arquitectura

La figura 2.1 muestra la arquitectura interna de PMCTrack v1.5, versión de partida para este proyecto. El módulo del kernel de PMCTrack implementa la mayor parte de la funcionalidad de PMCTrack y como se muestra en la figura 2.1, consta de varios componentes. Como se ha expuesto en el capitulo anterior, las aplicaciones y el usuario final pueden acceder a la funcionalidad de PMCTrack directamente desde el espacio de usuario. Estos componentes de espacio de usuario que ofrece PMCTrack, se comunican

con el núcleo independiente de la arquitectura del módulo del kernel de PMCTrack mediante un conjunto de entradas `/proc` de Linux exportadas por el propio módulo [1].

El módulo del kernel de PMCTrack, además de exponer los datos de los contadores de rendimiento de una aplicación a las herramientas del espacio de usuario, implementa un mecanismo sencillo para alimentar –de manera independiente de la arquitectura– con datos de monitorización del rendimiento por hilo a cualquier componente dentro del propio Linux.

Para recopilar los datos de los contadores de rendimiento por hilo, el módulo tiene que ser plenamente consciente de los eventos del planificador de hilos (p.ej. cambios de contexto, creación y terminación de hilos…) [1]. Dado que tanto el *core* del planificador de Linux como otras clases de planificadores se implementan por completo en el kernel, hacer que el módulo del kernel de PMCTrack sea consciente de los eventos relacionados con los hilos y de las peticiones del planificador del sistema operativo requiere algunas modificaciones en el propio kernel Linux. Estas modificaciones, denominadas API del kernel de PMCTrack en la figura 2.1, comprenden un conjunto de notificaciones emitidas desde el planificador del kernel al módulo de PMCTrack. Para recibir notificaciones clave, el núcleo independiente de la arquitectura del módulo del kernel de PMCTrack implementa una interfaz de operaciones (*callbacks*) que forman parte del parche de PMCTrack para el kernel Linux. La mayoría de estas notificaciones se activan sólo cuando el módulo del kernel de PMCTrack está cargado y el usuario, o el propio programador, está utilizando la herramienta para supervisar el rendimiento de una aplicación específica [1].

El módulo del kernel de PMCTrack además consta de módulos de monitorización. El propósito principal de estos módulos de monitorización es, como se ha mencionado anteriormente, proporcionar los datos de monitorización a componente del kernel Linux. Los módulos de monitorización también pueden exponer cualquier información de monitorización a los componentes del espacio de usuario de PMCTrack por medio de contadores virtuales. El soporte específico de la plataforma para los módulos de monitorización, se encapsula en un conjunto de PMU *Backends* (*Performance Monitoring Units*) dentro del módulo de PMCTrack.

Estos backends de PMCTrack realizan el acceso de bajo nivel a los contadores *hardware* de rendimiento –registros arquitectónicos– y a las unidades de monitorización del rendimiento (PMUs) del propio procesador. Las PMUs permiten la monitorización de métricas de la microarquitectura como son el número de ciclos, instrucciones retiradas, instrucciones ejecutadas, fallos sufridos en la caché L1, etc.

El acceso a los contadores *hardware* de rendimiento requiere acceder directamente a los registros especiales del *hardware*. Suele haber dos tipos de registros: los registros de configuración (que permiten iniciar y detener los contadores, elegir los eventos a monitorizar y configurar las interrupciones por desbordamiento) y los contadores en sí (que mantienen los recuentos de eventos actuales) [5].

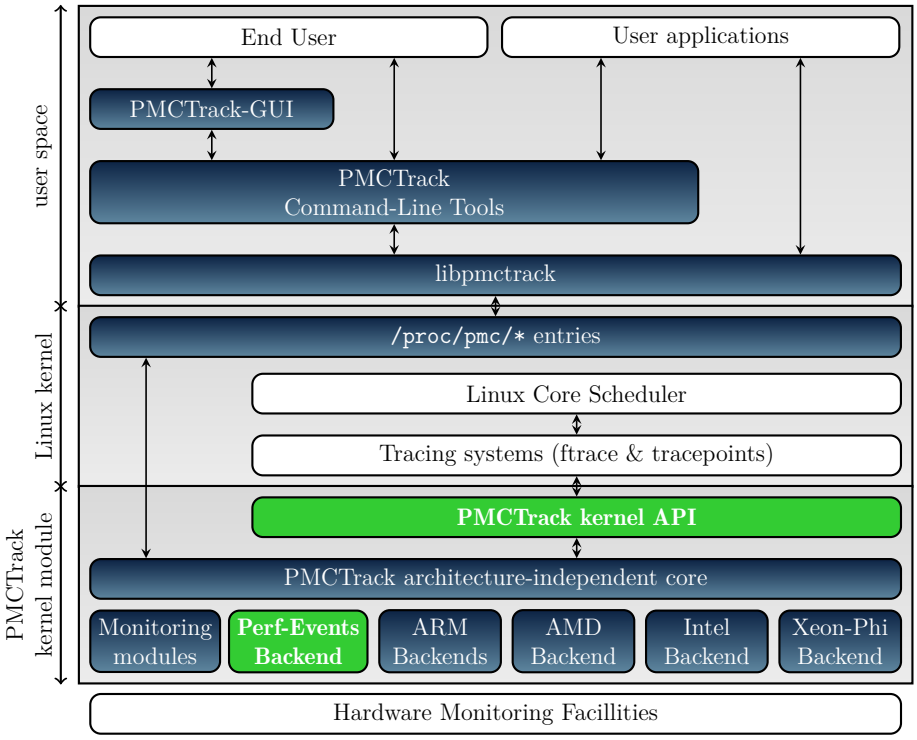


Figura 2.2: Arquitectura interna de PMCTrack v2.0

Los backends de PMCTrack deben realizar la lectura de los PMCs y PMUs disponibles en la plataforma, así como la traducción de las cadenas de configuración de los PMCs en estructuras de datos internas que puedan ser introducidas en los registros de configuración para cada plataforma en cuestión. Todo esto hace que sea necesario tener un backend específico por arquitectura.

Con la introducción de un nuevo backend experimental implementado con perf_events se expuso la posibilidad de abstraer el acceso a los registros *hardware* que deben realizar los backends de PMCTrack mediante el uso de perf_events. Probando así que es posible tener un backend genérico en PMCTrack.

Este trabajo de fin de grado surge de la búsqueda de la adaptación de la arquitectura de PMCTrack a entornos de producción. Para ello, uno de los objetivos es eliminar la necesidad del parche del kernel de PMCTrack para instalar la API de PMCTrack dentro del kernel Linux. Lograr esto, junto con la expansión de la funcionalidad del backend experimental de perf_events, supone que PMCTrack pase a tener la arquitectura interna representada en la figura 2.2. Esta nueva arquitectura de PMCTrack dispondría del soporte de la API del kernel de PMCTrack mediante el uso de sistemas de trazado, eliminando así el parche de PMCTrack.

2.3 Parche de PMCTrack (API del kernel de PMCTrack)

Como se comenta previamente, el parche de PMCTrack para el kernel Linux incluye la API de PMCTrack del kernel. Su inclusión requiere la adición de 2 nuevos ficheros fuente y, dependiendo de la arquitectura, alrededor de 20 líneas extra de código en los ficheros fuente del kernel Linux. Este parche de PMCTrack no es excesivamente complejo y modifica partes del código fuente de Linux que se han mantenido relativamente estables a lo largo de las versiones de Linux.

Para lograr la adaptación a una versión de PMCTrack sin parche del kernel se pueden seguir dos aproximaciones: crear un nuevo módulo del kernel para PMCTrack que incluya la adaptación entera de la API de PMCTrack, o por el contrario incluir la adaptación de la API de PMCTrack dentro del propio módulo de PMCTrack, junto con todo el resto de funcionalidades de PMCTrack, logrando así la arquitectura de la figura 2.2.

Por otro lado, para abordar su adaptación a una implementación que no dependa de un parche del kernel, se propone a continuación una división del parche de PMCTrack en términos de las funcionalidades que se implementan dentro de éste. Esta división se hace con la intención de aislar características que pueden requerir del uso de sistemas de trazado distintos para su adaptación, consiguiendo así un flujo de trabajo y estudio secuencial centrado en un grupo de sistemas de trazado reducido.

La división consta de tres partes: callbacks, datos de monitorización y resto del parche, explicadas a continuación.

2.3.1 Callbacks

Para recopilar los datos de los contadores de rendimiento por hilo, el módulo de PMCTrack tiene que ser plenamente consciente de los eventos de planificación de los hilos: cambios de contexto, creación/terminación de hilos etc. Para lograr esto PMCTrack incorpora al kernel una interfaz de callbacks llamada `pmc_ops_t` en dos nuevos ficheros fuente. Uno de ellos es `pmctrack.h`, creado en la ruta */include/linux* de las fuentes de Linux, que incorpora la definición de la siguiente interfaz:

```
typedef struct __pmc_ops {
    int     (*pmcs_alloc_per_thread_data)(unsigned long,struct task_struct*);
    void    (*pmcs_save_callback)(void*, int);
    void    (*pmcs_restore_callback)(void*, int);
    void    (*pmcs_tbs_tick)(void*, int);
    void    (*pmcs_exec_thread)(struct task_struct*);
    void    (*pmcs_free_per_thread_data)(struct task_struct*);
    void    (*pmcs_exit_thread)(struct task_struct*);
    int     (*pmcs_get_current_metric_value)(struct task_struct* task, int key,
                                             uint64_t* value);
} pmc_ops_t;
```

Esta interfaz de callbacks representa el conjunto de operaciones *wrapper* a incluir dentro

de las fuentes de Linux para su uso en PMCTrack. A continuación se explica una a una la funcionalidad de cada callback de PMCTrack instalada en el kernel Linux:

- **pmcs_alloc_per_thread_data**

  Se invoca cuando se crea un nuevo hilo. Esta callback se incorpora mediante el parche de PMCTrack dentro de la función `sched_fork()`.

- **pmcs_save_callback**

  Se invoca cuando un hilo sale de la CPU, es decir, cuando se produce un cambio de contexto saliente y se introduce dentro de la función `__schedule()`.

- **pmcs_restore_callback**

  Se invoca cuando un hilo entra en la CPU, se produce un cambio de contexto a dicha CPU. Esta se invoca desde dentro de la función `finish_task_switch()`.

- **pmcs_tbs_tick**

  Se invoca con cada tick del reloj en función de cada hilo. Esta se incorpora dentro de la función `scheduler_tick()`.

- **pmcs_exec_thread**

  Se invoca cuando un proceso invoca `exec()` y se introduce dentro de la función `sched_exec()`.

Las callbacks mencionadas hasta este punto junto con las funciones desde las que son invocadas se encuentran dentro del fichero */kernel/sched/core.c* del código fuente del kernel Linux.

- **pmcs_free_per_thread_data**

  Se invoca cuando un hilo sale del sistema. Esta callback se invoca desde dentro de la función `free_task()` en el fichero */kernel/fork.c* del código fuente de Linux.

- **pmcs_exit_thread**

  Se invoca cuando se libera el descriptor de un hilo. Esta callback se invoca desde dentro de la función `do_exit()` en el fichero */kernel/exit.c* del kernel.

Al contrario que el resto, la invocación de la última callback no se encuentra dentro del kernel Linux *vanilla*:

- **pmcs_get_current_metric_value**

  Esta callback es la encargada de exponer, de manera independiente a la arquitectura, las métricas de monitorización del rendimiento realizadas por el módulo de PMCTrack dentro del kernel. Será el programador el encargado de invocar a esta callback desde el código de los componentes del kernel en los que se quiera

aprovechar los datos de los contadores de rendimiento por hilo durante la ejecución de los mismos.

Aunque la invocación de estas callbacks se introdujeron en zonas del código que se consideraron más idóneas para su funcionalidad, dentro de las funciones de las que se quieren notificar, existe un margen para poder ser instaladas en otras zonas de código del kernel proporcionando el mismo resultado.

El otro fichero, `pmctrack.c` creado en la ruta */kernel* de las fuentes de Linux, contiene la implementación de las distintas llamadas de la interfaz dentro del kernel, que realizan la invocación de la funciones correspondientes del módulo del kernel de PMCTrack. Realmente la implementación de las callbacks que se introduce dentro del kernel Linux no es completa, sino que son funciones *wrapper* (envoltorio) que delegan la petición al módulo del kernel de PMCTrack. La implementación que se encuentra dentro del módulo del kernel de PMCTrack ya sí es la completa.

Como ejemplo, se muestra la codificación de la callback **pmcs_exec_thread** en el nuevo fichero `pmctrack.c` que se inserta en el kernel mediante el parche del núcleo de PMCTrack.

```
void pmcs_exec_thread(struct task_struct* tsk)
{
    pmc_ops_t* pmc_ops= NULL;

    if (!implementer)
        return;

    rcu_read_lock();

    pmc_ops=rcu_dereference(pmc_ops_mod);

    if(pmc_ops!=NULL && pmc_ops->pmcs_exec_thread!=NULL)
        pmc_ops->pmcs_exec_thread(tsk);

    rcu_read_unlock();
}
```

Esta callback al igual que el resto realiza las siguientes acciones:

- Comprueba que se haya instalado el módulo de PMCTrack dentro del kernel, denominado como `implementer`.
- Comprueba si dicho módulo de PMCTrack se encuentra en uso. Esto se consigue mediante complejos mecanismo de sincronización RCU (*Read, Copy, Update*) [26] y SpinLocks de lectura-escritura [27] dependiendo de la callback.
- Registra el alcance de dicha callback con sus argumentos para que sea ejecutada la implementación completa de la callback dentro del módulo de PMCTrack.

Estas comprobaciones han de realizarse ya que las callbacks son notificaciones que sólo

han de ejecutarse cuando el módulo del kernel de PMCTrack está cargado y PMCTrack está siendo utilizado.

La incorporación de la interfaz de callbacks de PMCTrack y su instalación dentro del código fuente de Linux, mediante el parche de PMCTrack, es una de las principales partes del parche a adaptar. Como se expone en los próximos capítulos, la existencia de mecanismos de trazado dentro del kernel Linux, que permiten la vinculación de callbacks a funciones dentro del kernel Linux en ejecución, presenta la posibilidad de no tener que instalar manualmente las callbacks de PMCTrack a través de un parche para el kernel.

2.3.2 Datos de monitorización

PMCTrack necesita mantener los datos de la monitorización de los contadores de rendimiento por proceso dentro del kernel. Esto se implementa en el parche de PMCTrack mediante la inclusión de dos campos nuevos en la estructura `task_struct` de cada tarea. El `task_struct` es la estructura que actúa como descriptor del proceso, conteniendo todo lo que el kernel puede necesitar saber sobre un proceso. Los dos nuevos campos son: una estructura de datos, `pmc`, y un nuevo flag, `prof_enabled`, ambos mantenidos por el módulo del kernel de PMCTrack.

```
struct task_struct {
    ...

#ifdef CONFIG_PMCTRACK
    void *pmc;                      /* Per-thread PMC-specific data */
    unsigned char prof_enabled;     /* This field must be one for the profiler
                                     * to be active in the current task */
#endif
    ...
};
```

Estos nuevos campos se encargan de indicar cuándo en una tarea la monitorización está activa y de proporcionar métricas de rendimiento de alto nivel u otra información de tiempo de ejecución que es potencialmente expuesta por el *hardware* (a través de PMCs o por otros medios) [1].

El campo `pmc`, aunque realmente se emplea para apuntar a una estructura de datos de tipo `pmon_prof_t`, se define como tipo `void`. Esto se debe a que la declaración del tipo de estructura `pmon_prof_t` se encuentra dentro del propio módulo del kernel de PMCTrack y ésta está sólo disponible cuando el módulo se carga en el kernel. El campo `prof_enabled` tomará valor de 1 o 0 dependiendo de si la monitorización esta activa o no respectivamente, para el hilo en cuestión.

La necesidad de mantener campos extra para datos por proceso no resulta extraño de considerar para otras herramientas, como puede ser `perf`, por lo que un primer enfoque podría ser la creación de un parche para el kernel –con los campos de PMCTrack dentro del `task_struct`– y su posterior solicitud de inclusión en el Linux *mainline*. Como se

acaba de comentar, `perf` es capaz de mantener datos de monitorización por proceso, lo que expone la posibilidad de estudio del subsistema perf_events para encontrar una posible forma de mantener los datos de monitorización de PMCTrack por hilo haciendo uso de perf_events. Ambos enfoques son expuestos en los próximos capítulos.

2.3.3 Resto del parche

El resto del parche incluye características adicionales y la definición de variables que no proponen plantear el uso de sistemas de trazado para su adaptación.

El parche de PMCTrack incorpora dos llamadas nuevas para poder registrar la instalación del módulo PMCTrack dentro del kernel Linux. La adaptación de estas funciones es trivial, ya sea desde dentro del kernel o desde un módulo, la implementación del registro o su eliminación siempre se es similar.

```
int register_pmc_module(pmc_ops_t* pmc_ops_module, struct module* module);
int unregister_pmc_module(pmc_ops_t* pmc_ops_module, struct module* module);
```

Por último PMCTrack necesita código especifico para poder recuperar el `task_struct` asociado a una tarea dado su pid (*process identifier*). Se ha de tener en cuenta que existe la función `find_process_by_pid()` de Linux –una función privada del kernel y no accesible desde los módulos cargables– la cual devuelve el `task_struct` asociado a un identificador de proceso (pid). Por tanto, para ello se cambia, con el parche de PMCTrack, la función `find_process_by_pid()` de Linux a no `static`, eliminado la restricción de un alcance limitado a su fichero objeto. Después se exporta `find_process_by_pid()` para que pueda ser usada desde el módulo de PMCTrack, como se muestra a continuación:

```
#ifdef CONFIG_PMCTRACK
struct task_struct *find_process_by_pid(pid_t pid)
#else
static struct task_struct *find_process_by_pid(pid_t pid)
#endif
{
    return pid ? find_task_by_vpid(pid) : current;
}
#ifdef CONFIG_PMCTRACK
EXPORT_SYMBOL_GPL(find_process_by_pid);
#endif
```

Resulta extraño que no se haya considerado el ofrecer conseguir el `task_struct` de un proceso a través de su identificador dentro de módulos para el kernel. Por ello se puede plantear la posibilidad de proponer un parche con la exportación de `find_process_by_pid()` a la comunidad de Linux.

Capítulo 3

Sistemas de trazado en el kernel Linux

En este capítulo se expone los distintos tipos de trazado (*tracing*) que existen dentro del Linux *mainline*. También se explican los sistemas de trazado que implementan o utilizan dichos tipos de trazado. Todo ello orientado a su posible aplicación en este proyecto.

El capítulo se estructura de la siguientes forma. En la **sección 3.1** se expone un breve recorrido sobre la historia de los sistemas de trazado, desde su aparición hasta el momento de inicio de este proyecto. En la **sección 3.2** se plantea como abordar el estudio de los sistemas de trazado. En la **sección 3.3** se describe en detalle cómo implementan el trazado dentro de Linux los sistemas de trazado **tracepoints**, **ftrace** y **Kprobes**. En la **sección 3.4** se explican las funcionalidades que ofrecen los sistemas de trazado, haciendo una relación a su posible aplicación en la adaptación de la API de PMCTrack. Por último, en la **sección 3.5** se explica el subsistema de eventos perf_events como sistema de rastreo y herramienta de monitorización del rendimiento, para su uso en la adaptación del mantenimiento de datos de monitorización por hilo para PMCTrack.

3.1 Contexto histórico

Dado que la API de PMCTrack para el kernel consiste en la instalación de su interfaz de callbacks dentro de funciones del núcleo del planificador de Linux, la existencia de sistemas de trazado dentro del propio kernel Linux –que permiten asociar funciones dadas por el usuario a funciones dentro del código fuente del kernel– plantea la posibilidad de alcanzar el deseado estado de PMCTrack. Estado en el que no se requiera de un parche del kernel para la instalación de la API de PMCTrack, siendo así PMCTrack compatible con entornos de producción.

La necesidad de poder conocer que sucede dentro del código fuente en tiempo de ejecución ha estado siempre presente. No fue hasta finales de la década de 1990 cuando empezaron a aparecer las primeras implementaciones que ofrecían trazado sobre el código fuente del kernel Linux sin necesidad de la creación, por parte de cada desarrollador, de un sistema

de trazado propio. A pesar de todo, en esta época el trazado dentro del código fuente del kernel Linux aún no era considerado una necesidad lo suficientemente importante para los desarrolladores como para justificar la inclusión de complejos sistemas de trazado dentro de Linux *mainline*. Se prefería invertir el tiempo, para el trazado ocasional, en escribir un trazador a partir de parches de trazado preexistentes, adaptados a problemas específicos [28].

Aún así se empezaron a plantear los primeros sistemas de trazado dentro del código fuente del kernel Linux, por ejemplo logdev (1998) [29], DProbes (2000) [30] y Linux Trace Toolkit (LTT) (1999) [31]. LTT fue el primer intento real de integrar el trazado en el Linux *mainline*. A Ingo Molnar y Linus Torvalds no les gustó cómo se implementó porque introducía una carga constante en el código fuente del kernel no justificada ya que, en su opinión, el 99% del desarrollo no requería nunca de trazado. Ésto hizo que fuera rechazada la incorporación de LTT dentro de las fuentes del kernel *mainline* [28].

En 2004 apareció Kprobes inspirado en DProbes, un sistema de trazado dinámico simple y ligero del kernel, que proporcionó facilidad para ejecutar un manejador definido por el usuario cuando se alcanza un punto de traza instalado dinámicamente en la fuentes de código de Linux [32]. Kprobes se basa en la instalación de instrucciones de interrupción para crear los puntos de traza dinámicos, por lo que puede añadir una sobrecarga considerable.

En 2004 también apareció *latency tracer* (un parche que introdujo *preempt-realtime*) [33] que junto con logdev se combinaron formando el primer sistemas de trazado genérico incluido en las fuentes del kernel Linux, llamado **ftrace**. Ftrace fue incluido en el kernel Linux *mainline* 2.6.27 en 2008, introduciendo un trazado dinámico de funciones dentro del kernel con una sobrecarga muchísimo menor a kprobes.

En 2007 se añadió dentro del kernel Linux *mainline* los Kernel Markers, un tipo de trazado estático en respuesta a la sobrecarga que suponía usar Kprobes y a la necesidad de un trazado que se pudiese mantener estable entre versiones del kernel [34]. Para resolver problemas relacionados con los Kernel Markers, se implementó una versión más sencilla y segura denominada **tracepoints**, incluidos en 2008 en la rama principal del kernel Linux [35]. Los tracepoints se han mantenido hasta el día de hoy dentro de las fuentes de Linux y proporcionan puntos de traza estáticos incorporados por los desarrolladores del núcleo dentro del código fuente.

A lo largo de los años fueron apareciendo nuevas implementaciones de tecnologías de trazado en Linux. Algunas como SystemTap (2009) nunca llegaron a ser incorporados en la rama principal del kernel Linux [36]. Otras como BPF (Berkeley Packet Filter) 2011 si que han visto la luz dentro del código fuente del kernel Linux [37]. BPF proporcionó la capacidad de analizar el tráfico de la red y fue posteriormente sustituida por eBPF (Extended BPF) en 2014 [37], [38].

En la actualidad los sistemas de trazado como ftrace, tracepoints o kprobes han sido ampliamente usados por la comunidad de Linux y se han establecido como estándares

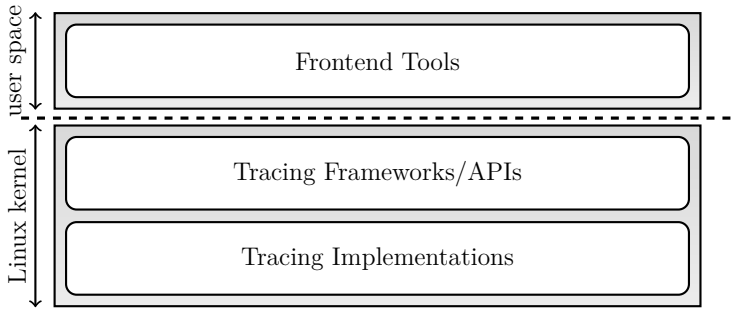


Figura 3.1: Trazado en Linux

de trazado en Linux, gracias a que han alcanzado una enorme robustez y madurez.

Ftrace se usa para aplicaciones tan críticas como es el framework de parches en vivo dentro del kernel (*live patching*), incluido en las fuentes de la rama principal del kernel Linux [11].

Los Tracepoints se emplean ampliamente por numerosas herramientas de trazado incluyendo ftrace e incluso perf_events, y se van añadiendo cada vez más dentro del código fuente del kernel Linux.

Todo ésto expone un contexto en el que es una realidad el considerar posible la utilización de estos sistemas de trazado para alcanzar una versión de PMCTrack compatible con entornos de producción. La arquitectura interna de dicha versión de PMCTrack a alcanzar, se encuentra representada en la figura 2.2 de la sección 2.2.

3.2 Introducción a los sistemas de trazado en Linux

El estudio de los sistemas de trazado en Linux puede resultar confuso por la gran variedad de sistemas que se han ido creando y que existen actualmente (strace, ltrace, kprobes, tracepoints, uprobes, ftrace, perf, eBPF, etc) y su distinta, pero en algunos casos similar, funcionalidad [29], [39]. En un intento de clarificar dicho estudio se puede hacer una división de las funcionalidades que ofrecen los sistemas de trazado en Linux (ilustrado en la figura 3.1) en tres clases [40]:

- Implementaciones de trazado, centrado en exponer las fuentes de datos, de dónde provienen los datos de trazado.
- *Frameworks* o APIs de trazado, mecanismos para recolectar datos de esas fuentes.
- Herramientas *frontend* de trazado, la herramienta con la que se interactúa para recolectar/analizar datos.

Un sistema de trazado normalmente no implementa sólo una de estas funcionalidades si no una combinación entre ellas. Con relación a este proyecto, las funcionalidades de principal importancia de estudio son las implementaciones de trazado y los distintos

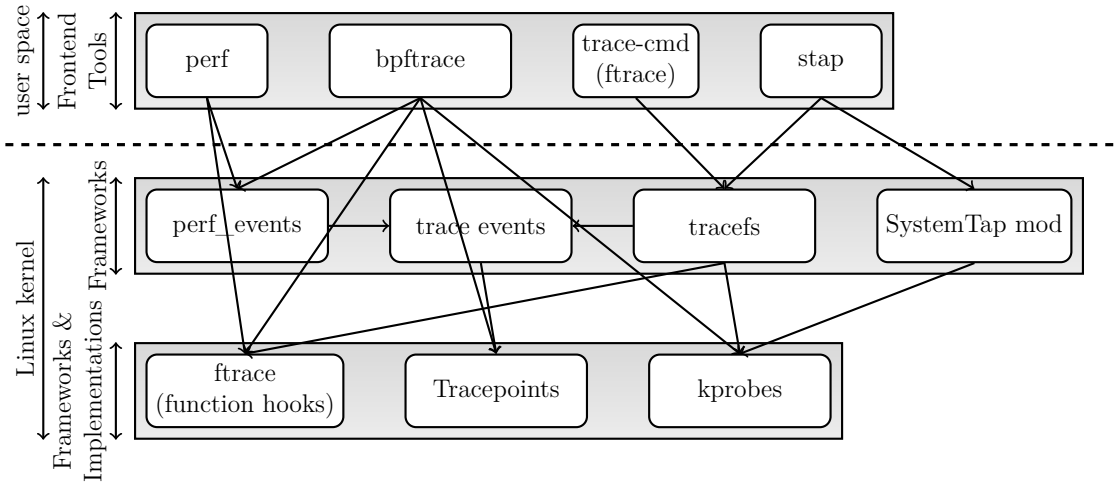


Figura 3.2: Interrelaciones entre los sistemas de trazado en Linux relevantes para PMCTrack

mecanismos para recolectar datos de dichas implementaciones que existen dentro del propio kernel Linux.

Aún con la extensa colección de sistemas de trazado, éstos no pretenden ser competencia entre sí, cada uno tiene sus puntos fuertes y débiles. Todos son de código abierto y utilizan, o pueden utilizar, características de otros sistemas de trazado [29]. En la figura 3.2 se ilustra la compartición y utilización entre sistemas de trazado que se han considerado más relevantes en el ámbito de este proyecto.

Debido al amplio abanico de sistemas de trazado que existen es necesario concretar el uso que se pretende dar a dichos sistemas, con el objetivo de poder centrar el estudio en aquellos que tienen las características requeridas para su aplicación en este proyecto. El uso principal que se persigue consiste en poder invocar una callback cuando cierta función específica dentro del kernel Linux sea invocada, ya sea dentro de ésta misma o justo antes de ejecutase. Ésto tiene que poder realizarse de la manera más eficiente posible, reduciendo al mínimo la sobrecarga que supone la instalación de dicha callback dentro del kernel. Es necesario que dicho sistema de trazado a usar esté presente en versiones posteriores a la actual del kernel Linux, en concreto para PMCTrack al menos desde la versión 4.1, y que se tenga cierta garantía de que la comunidad de Linux está comprometida a su mantenimiento para futuras versiones de Linux. Todos estos requisitos implican que las implementaciones de trazado a considerar tengan que ser suficientemente maduras y que se hayan probado y establecido su eficiencia dentro del kernel Linux. También supone el uso de *frameworks* de trazado robustos que ofrezcan los mecanismos necesarios para satisfacer este uso dentro del kernel Linux.

3.3 Implementaciones de trazado dentro del kernel Linux

Una vez establecidos los requisitos mínimos de los sistemas de trazado a escoger para este proyecto, es importante empezar el estudio por las implementaciones de trazado que existen. El trazado dentro del kernel Linux se puede realizar con distintas implementaciones para ofrecer las fuentes de datos; dichas implementaciones son la base para el resto de funcionalidades de los sistemas de trazado. Dentro de las implementaciones de trazado se hace una distinción clara en dos tipos: estático y dinámico.

3.3.1 Trazado estático

El trazado estático se implementa mediante puntos de traza estáticos añadidos en el código fuente. Tienen una baja carga de procesamiento, pero el código en el que se puede trazar es limitado y se define en tiempo de compilación. Estos puntos de traza estáticos representan eventos importantes que pueden ser tomados en conjunto con otros para proporcionar una idea general de lo que está sucediendo dentro del sistema [41]. Un punto de traza estático se compone de un nombre junto a metadatos, que son elementos específicos de este evento. Cuando se alcanza dicho punto de traza en un momento determinado, una herramienta puede capturar el evento con todos sus metadatos [42]:

- El tiempo, con precisión de nanosegundos.
- Qué proceso lo ejecutó.
- En qué CPU se ejecutó.
- También existe la posibilidad de capturar las cadenas de llamadas, es decir, la secuencia de llamadas a funciones anidadas que conducen al evento.

La motivación inicial para crear una infraestructura de instrumentación estática es la gran sobrecarga de rendimiento inducida por mecanismos de instrumentación dinámica como Kprobes, que inyecta instrucciones de interrupción. El uso de trazado estático como implementación para la adaptación de las callbacks de PMCTrack es el ideal en términos de eficiencia en sobrecarga.

Existen múltiples implementaciones de trazado estático, USDT (*User Statically-Defined Tracing*) [43], tracepoints, lttng-ust [44], etc. La de mayor interés para este proyecto son los tracepoints por la funcionalidad que ofrecen, por su aceptación por la comunidad de Linux y por la inclusión en la distribución del kernel Linux *mainline* desde la versión 2.6.28. Ésto implica que esta instrumentación estática puede sobrevivir más fácilmente a los cambios en el código fuente.

Los **tracepoints** implementan el trazado estático mediante la creación de "funciones gancho", tracepoints, que se definen en una serie de ficheros de cabecera en el directorio */include/trace/events* del código fuente del kernel. Esta definición se realiza a través de varias macros proporcionadas por su infraestructura. Estos tracepoints se instalan en el código del kernel Linux y proporcionan un "gancho" para llamar a una función (callback) que se puede proporcionar en tiempo de ejecución, a la cual se le permiten

pasar y modificar un número determinado de parámetros establecidos en la definición del tracepoint. Un tracepoint puede estar "activado" (una función callback está conectado a él) o "desactivado" (no se le conecta ninguna función callback) [8]. Cuando un tracepoint está "desactivado", no tiene ningún efecto, salvo que añade una pequeña penalización de tiempo por la comprobación de una condición de salto, y de espacio en memoria, pues añade unos pocos bytes para la llamada a la función al final de la función instrumentada y agrega además una estructura de datos en una sección separada. Por el contrario cuando el tracepoint está "activado", la función proporcionada es llamada cada vez que se ejecuta el tracepoint, en el mismo contexto de ejecución en el cual se ejecuta dicho tracepoint. Cuando la función proporcionada termina su ejecución se vuelve, continuando desde el sitio donde se encuentra el tracepoint.

Los tracepoints han sido añadidos en distintas localizaciones del código fuente por los desarrolladores del kernel Linux para supervisar los subsistemas del kernel, como el planificador, administración de energía, interrupciones, red, *gpio*, etc [45]. Aun así la limitación de la cantidad de puntos de traza que ofrecen los tracepoints y su posible inexistencia en los puntos de interés para PMCTrack hace que se tengan que plantear también el uso de sistemas de trazado dinámicos. Para hacer uso de los tracepoints es necesaria la compilación de kernel Linux con el símbolo de preprocesado `CONFIG_TRACEPOINTS` activo, que ofrece también una API de trazado para los tracepoints. En la siguiente sección 3.4.1 ésto se expondrán en más detalle.

3.3.2 Trazado dinámico

El trazado dinámico se implementa mediante puntos de traza inyectados dinámicamente en el código, lo que permite definir en tiempo de ejecución el código a trazar. Esto implica que el rango de código fuente en el que se puede hacer trazado es mucho más amplio, pero conlleva una cierta carga de procesamiento que varia según la implementación del trazado dinámico. Este tipo de trazado es fácil de mantener ya que no hay que modificar el código fuente donde se pretende trazar. Las dos principales implementaciones de trazado dinámico dentro del kernel Linux son **ftrace** y **kprobes**. Estos sistemas de trazado han sido aceptados y mantenidos por la comunidad de Linux en la distribución oficial de Linux desde las primeras versiones 2.6 y han logrado un alto grado de madurez.

3.3.2.1 Ftrace (*Function Hook*)

Ftrace implementa "ganchos de función" que permiten la instalación de callbacks en la mayoría de las funciones del kernel. Internamente, ftrace se basa en los mecanismos de *profiling* de `gcc` (*GNU Compiler Collection*) [46] para añadir instrucciones de máquina en el prólogo de las versiones compiladas de la gran mayoría de las funciones del código fuente del kernel Linux. Estas instrucciones redirigen la ejecución de las funciones a los trampolines de ftrace. Ftrace intercambia las instrucciones de "punto de entrada" creadas por `gcc` por NOPs cuando el kernel arranca. Éstas pueden ser alteradas por

ftrace posteriormente en tiempo de ejecución entre NOPs y saltos reales a los trampolines de trazado [47].

Cuando el kernel Linux se compila con soporte de ftrace, el compilador `gcc` hace uso de su opción `-pg -mfentry` la cual inserta la llamada especial `mcount` o `__fentry__`, en función de la arquitectura, en el prólogo de cada función del código fuente del kernel. Esta inserción no se realiza en las funciones definidas como `inline` y algunas funciones especiales, marcadas con `notrace` para evitar el trazado sobre dichas funciones. Tomando como ejemplo la función del planificador del kernel para ilustrar todo el proceso, se parte de la siguiente implementación:

```
__visible void __sched schedule(void)
{
    struct task_struct *tsk = current;
    sched_submit_work(tsk);
    ...
}
```

El prólogo de la llamada de la función desensamblada para el kernel compilado con las opciones `-pg -mfentry` tiene la siguiente codificación:

```
<schedule>:
    e8 1b d0 1e 00          callq ffffffff81c01930 <__fentry__>
    ffffffff81a14911:  R_X86_64_PLT32     __fentry__-0x4
    53                      push    %rbx
    65 48 8b 1c 25 00 61    mov     %gs:0x16100,%rbx
    01 00
    ffffffff81a1491b:  R_X86_64_32S  current_task
    ...
```

Esta llamada especial `__fentry__` funciona como un trampolín para saltar a código C y se invoca desde todas las funciones del kernel en la que ha sido instalada. Esto implica que si no se pretende hacer uso de ella, esta llamada solo retornará al ser llamada. Este proceso de llamar a `__fentry__` y retornar en todas las funciones del kernel añade una sobrecarga del 13% [48]. Es por ésto que ftrace convierte todas las llamadas `__fentry__` por NOPs en el arranque del kernel Linux. No obstante, para poder deshacer ésto posteriormente se requiere conocer donde se encuentran para su posterior trazado, por lo que es necesario guardar las direcciones a dichas llamadas.

```
<schedule>:
    0f 1f 44 00 00          nop
    53                      push    %rbx
    65 48 8b 1c 25 00 61    mov     %gs:0x16100,%rbx
    01 00 ffffffff81a1491b: R_X86_64_32S current_task
    ...
```

En tiempo de compilación del kernel, sí se construye con soporte de ftrace, se leen los ficheros objeto uno a uno encontrando todas las llamadas `__fentry__`, se crea una tabla con cada localización y se enlaza al final de cada fichero objeto, en una nueva sección llamada `__mcount_loc`. En esta sección se pasa a tener todas las direcciones

a las llamadas `__fentry__` dentro del fichero objeto. Ésto lo realiza `gcc` con la opción `-mrecord-mcount`. Este proceso junto con los que continúan se ven representados de manera gráfica en la figura 3.3.

También en tiempo de compilación, cuando el enlazador crea el archivo vmlinux –la imagen del kernel residente creada por el enlazado de todos los ficheros objeto del kernel Linux– el enlazador crea dos variables `__start_mcount_loc` y `__stop_mcount_loc`. Estas variables definen el lugar en que enlaza cada sección de todos los ficheros de cabecera con todas las direcciones a las llamadas `__fentry__`; una vez enlazadas, estas direcciones se convierten a direcciones físicas. Actualmente el recorrido de esta sección en tiempo de arranque por ftrace para convertir cada dirección por instrucciones NOPs, no es necesario ya que ésto lo realiza automáticamente `gcc` con la opción `-mnop-mcount`.

En tiempo de arranque, tras la sustitución por NOPs, esta sección con todas las direcciones a las llamadas `__fentry__` se elimina de vmlinux. El motivo de su eliminación es porque el array de direcciones construido dentro de vmlinux no es suficiente, se necesita tener estados de trazado para cada una de estas localizaciones. Cada dirección es copiada dentro de la estructura `dyn_ftrace` y asignada en nuevos grupos de páginas creados, donde son ordenadas por la dirección. En el sistema estas páginas pueden ser alrededor de 150 y ocupar un poco más de 0.5MBytes.

```
struct dyn_ftrace {
    unsigned long           ip;     /* address of __fentry__ call-site */
    unsigned long           flags;
    struct dyn_arch_ftrace  arch;
};
```

Los estados de trazado de cada llamada `__fentry__` vienen representados por el campo `flags` –de su estructura `dyn_ftrace`–, en el que se usan distintos bits para representar los distintos estados en que se puede encontrar: el número de callbacks registradas, si se está trazando o no la función, si se necesita guardar los registros para la callback, etc. Todo este proceso, desde la recopilación de las localizaciones de las llamadas `__fentry__` hasta su almacenamiento en nuevas páginas de ftrace, se representa gráficamente en la figura 3.3.

Estas nuevas instrucciones NOPs, que han sustituido a las llamadas `__fentry__`, han de ser modificadas en tiempo de ejecución para volver a instalar la llamada `__fentry__` en aquellas funciones en las que se activa el trazado en su estado. Esto presenta un problema en sistemas SMP (*Symmetric multiprocessing*); otra cpu puede estar ejecutando el código que se pretende modificar. En arquitecturas como x86, las instrucciones no son uniformes por lo que no hay ninguna garantía de donde se va a emplazar la instrucción en tiempo de ejecución. La manera de solventar este problema consiste en el uso de *breakpoints*. Se instala un *breakpoint* en la instrucción NOP (int3 en x86_64) y se sincronizan todas las CPUs y la memoria enviando una IPI (*Inter-Processor Interrupt*) a todas las CPUs.

```
<schedule>:
    <cc>0f 1f 44 00 00      <int3>nop
```

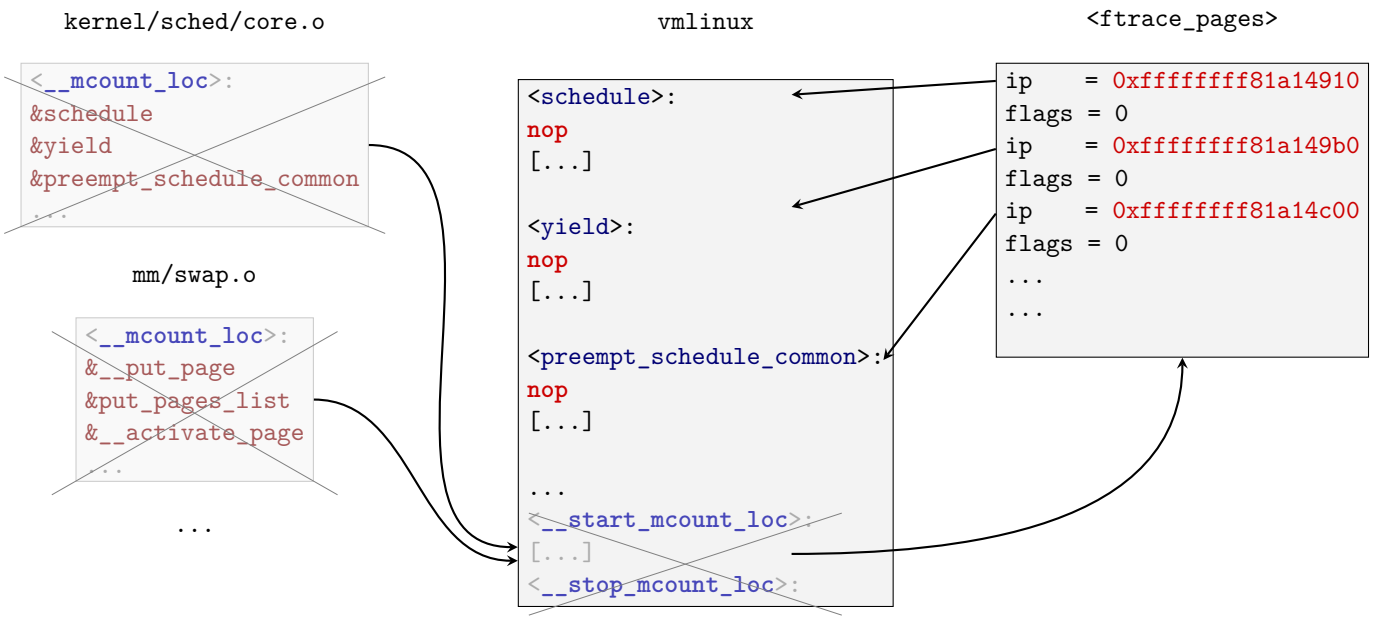


Figura 3.3: Creación de las páginas ftrace

```
    53                      push   %rbx
    65 48 8b 1c 25 00 61    mov    %gs:0x16100,%rbx
    01 00 ffffffff81a1491b: R_X86_64_32S current_task
    ...
```

Cuando se alcanza el *breakpoint*, se llama a la rutina de interrupción que llama al manejador de interrupción de ftrace. Éste modifica el contador de programa (IP en x86) para saltar sobre la instrucción NOP y una vez saltada se modifica por la llamada a `__fentry__`:

```
<schedule>:
    <cc>1b d0 1e 00         <int3>callq ffffffff81c01930 <__fentry__>
    53                      push   %rbx
    65 48 8b 1c 25 00 61    mov    %gs:0x16100,%rbx
    01 00 ffffffff81a1491b: R_X86_64_32S current_task
    ...
```

Finalmente se vuelven a sincronizar todas las CPUs de nuevo y se elimina el *breakpoint*:

```
<schedule>:
    e8 1b d0 1e 00          callq  ffffffff81c01930 <__fentry__>
    53                      push   %rbx
    65 48 8b 1c 25 00 61    mov    %gs:0x16100,%rbx
    01 00 ffffffff81a1491b: R_X86_64_32S current_task
    ...
```

Algunas arquitecturas ni siquiera necesitan hacer la sincronización, y pueden simplemente

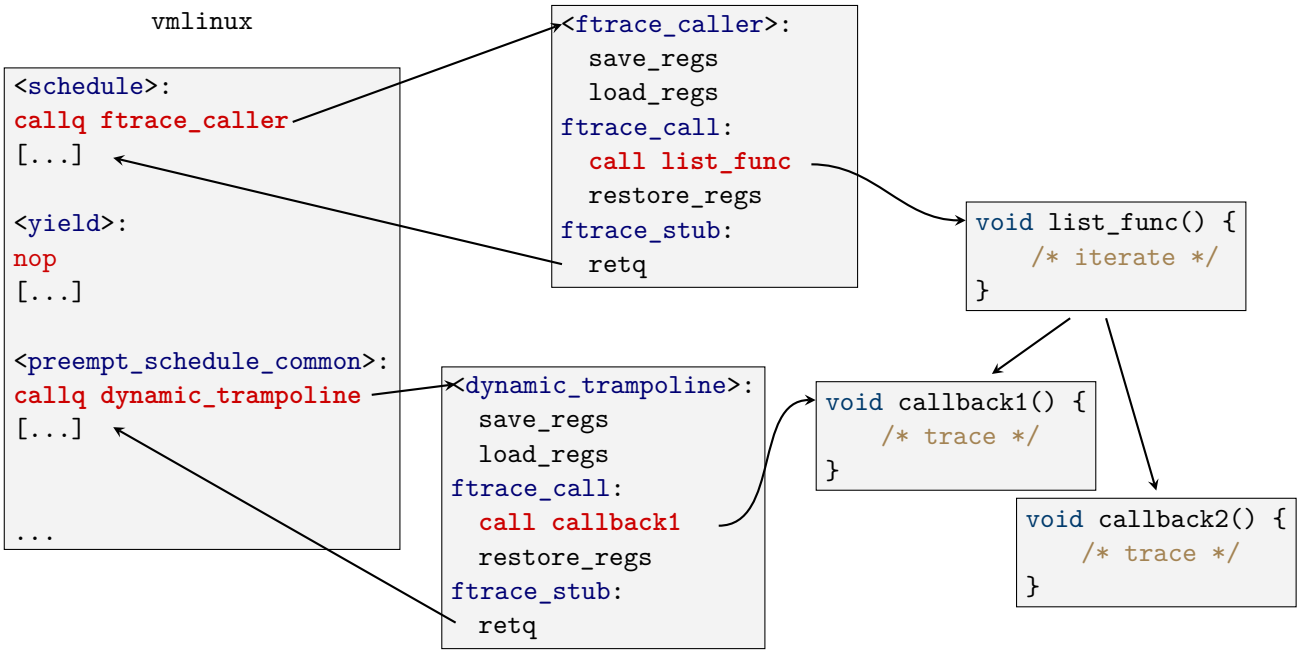


Figura 3.4: Representación del funcionamiento de los trampolines de ftrace cuando se registran callbacks

colocar el nuevo código sobre el NOP sin ningún problema con otras CPUs que lo ejecuten al mismo tiempo [10].

Llegado a este punto la llamada `__fentry__` redirige la ejecución al trampolín de ftrace, `ftrace_caller` (figura 3.4). Por defecto el trampolín simplemente guarda los registros y los carga, si así se indica en su estado de trazado, y hace la llamada `ftrace_stub`, función que no hace nada, y retorna. Esto se hace porque es mucho mas simple modificar dinámicamente el trampolín de forma que cuando se añade una función –callback a ejecutar cuando se alcanza la función trazada– `ftrace_stub` pasa a ser sustituida por la llamada `list_func`. Esta llamada corresponde a una función de ftrace que itera por cada callback proporcionada [45]. Ftrace también implementa trampolines dinámicos, `dynamic_trampoline`, para las situaciones en que se instalan callbacks iguales en distintas funciones del kernel. En estos trampolines la llamada `ftrace_stub` es directamente remplazada por la llamada a función de la callback proporcionada. El proceso de los trampolines de ftrace se representan en la figura 3.4.

Para poder hacer uso de la implementación de trazado de ftrace es necesaria la compilación del kernel Linux con los flags `CONFIG_FTRACE`, `CONFIG_DYNAMIC_FTRACE_WITH_REGS` y `CONFIG_FUNCTION_TRACER` activos. El sistema funcionará sin prácticamente ninguna sobrecarga cuando no se realice trazado de funciones. Esta compilación incluye el propio framework de trazado de ftrace en el kernel, que se describe en la próxima sección 3.3.2.

Ftrace implementa un trazado dinámico con el objetivo de tener la menor sobrecarga posible. Hace uso de *breakpoints* sólo la primera vez que se registra el trazado sobre una función del kernel Linux. Esto hace que ftrace sea una implementación de trazado prioritaria a usar para este proyecto frente a otras implementaciones dinámicas que presentan mucha más sobrecarga.

La implementación de ftrace es compleja y extensa y no ha podido ser recopilada en su totalidad en esta sección. Para completar el conocimiento sobre la implementación de ftrace se recomienda consultar una excelente charla realizada por Steven Rostedt, uno de los creadores de ftrace, en la conferencia *Kernel Recipes* de 2019 [49].

3.3.2.2 Kprobes (*Kernel Probes*)

Kprobes implementa una "sonda del kernel" dinámica, kprobe, capaz de ser instalada en prácticamente cualquier función del código fuente del kernel Linux. La idea general de la instalación de una kprobe consiste en la capacidad de insertar un punto de interrupción en casi cualquier dirección de código del kernel en tiempo de ejecución, la capacidad de invocación a un manejador dado para dicho punto de interrupción cada vez que sea alcanzado y la posterior continuación de la ejecución.

Cuando se registra una kprobe, se hace una copia de la instrucción trazada y se reemplaza el primer byte(s) de la instrucción trazada con una instrucción de interrupción (por ejemplo, int3 en i386 y x86_64) [9]. Esto se muestra en la figura 3.5.

Cuando se alcanza la instrucción de interrupción en una CPU, se produce una "trampa", los registros de la CPU se guardan y el control pasa a Kprobes a través del mecanismo `notifier_call_chain` (figura 3.5). Kprobes ejecuta la rutina de manejo "`pre_handler`" asociado con la kprobe, pasando al manejador las direcciones de la estructura de la kprobe y los registros del kernel guardados [9]. Estas rutinas de manejo proporcionadas a la kprobe se ejecutan como extensión del manejador de interrupciones del sistema. Debido a este diseño, las kprobes son capaces de implantarse en los entornos más hostiles: en tiempo de interrupción, entre cambios de contexto, en rutas de código habilitadas para SMP, etc [50].

A continuación, Kprobes va recorriendo paso a paso su copia de la instrucción trazada. Resultaría más sencillo realizar un recorrido paso a paso de la instrucción real en su lugar, pero entonces Kprobes tendría que eliminar temporalmente la instrucción de interrupción. Ésto abriría una pequeña ventana de tiempo en la que otra CPU podría pasar por delante del punto de traza [9].

Después de que la instrucción es recorrida paso a paso, Kprobes ejecuta el manejador "`post_handler`" que está asociado con la kprobe, si se ha especificado alguno. La ejecución continúa entonces con la instrucción que sigue al punto de traza [9]. Todo este flujo de control de Kprobes se puede ver representado gráficamente en la figura 3.5.

Kprobes es un sistema de trazado maduro y bien documentado, por lo que su sobrecarga

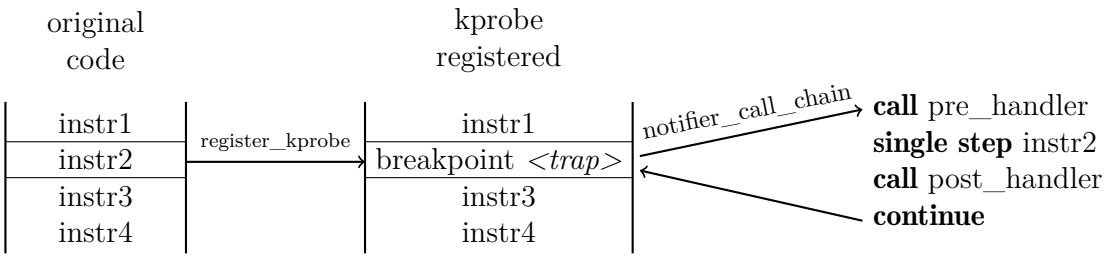


Figura 3.5: Flujo de control de Kprobes

es bien conocida y ha sido ampliamente estudiada. En una CPU típica en uso en 2005, el procesamiento del alcance de una kprobe lleva entre 0.5 y 1 microsegundos [9]. Kprobes ofrece distintos métodos para lograr una optimización de la sobrecarga sujetos a características especificas de aspectos como la arquitectura del sistema, la instrucción en la que es instalado, las especificaciones del sistema, la configuración del sistema operativo, etc. Entre ellos se encuentra la optimización del salto de Kprobes, que intenta reducir la sobrecarga de la kprobe utilizando una instrucción de salto en lugar de una instrucción de interrupción en cada punto de traza [9]. Normalmente, cuando se alcance un kprobe optimizado pasa a tardar entre 0.07 y 0.1 microsegundos en procesarse [9]. Esta sobrecarga y otros inconvenientes, que se exponen en el próximo apartado 3.4.3, hacen que no se haya optado por usar este sistema de trazado en el proyecto.

3.4 Frameworks/APIs de trazado en el kernel Linux de interés para PMCTrack

Una vez descritas las distintas implementaciones de trazado dentro del kernel Linux y establecido cuáles pueden resultar de interés para ser usadas junto con un framework de trazado para la adaptación de las callback de PMCTrack, es momento de exponer qué frameworks o APIs han sido elegidos a usar en este proyecto. Se ha de tener en cuenta que las implementaciones de trazado discutidas en la sección anterior, ofrecen su propio framework o API dentro del kernel Linux.

3.4.1 Tracepoints

Como se ha mencionado al hablar de trazado estático, los tracepoints son puntos estáticos de interés colocados en ubicaciones estratégicas del código del kernel Linux por los desarrolladores del mismo [42]. Un tracepoint colocado en el código proporciona un gancho para llamar a una función, que se puede proporcionar en tiempo de ejecución, a la que se le pasa un conjunto único de parámetros definidos por cada tracepoint.

Los tracepoints están inspirados en los Kernel Markers, que eran una implementación de puntos de traza estáticos para el kernel Linux. Creaban un mecanismo de apoyo del kernel para la utilización de trazado estático del código fuente del kernel Linux.

Este mecanismo permitía a herramientas especiales como LTT o SystemTap trazar la información expuesta por estos puntos de traza [34], [35]. Los Kernel Markers fueron una derivación del conjunto de parches externos LTT (Linux Trace Toolkit) que hoy se conoce como LTTng, un framework de trazado de código abierto para Linux [51], [52]. Los Kernel Markers incorporaron una API dentro del kernel que permitía la creación de puntos de traza estáticos y el enganche de funciones a ellos sin necesidad de un trazado dinámico [34]. Se incluyeron en la versión 2.6.24 del kernel Linux en 2008 y posteriormente, para resolver los problemas relacionados con los Kernel Markers, Mathieu Desnoyers, su creador, implementó una versión más simple y más segura de los puntos de traza estáticos llamada tracepoints. Los tracepoints fueron incorporados en la versión 2.6.28 del kernel Linux en 2008. A partir de entonces, los Kernel Markers se eliminaron lentamente de las fuentes del kernel y finalmente se eliminaron por completo en la versión 2.6.32 [35].

Los tracepoints ofrecen una API de trazado con mecanismos para utilizar su propia implementación de trazado estático dentro del kernel Linux. Esta API de tracepoints existe desde la inclusión de los tracepoints en el kernel Linux (versión 2.6.28) y se encuentra definida en el fichero de cabecera */include/linux/tracepoints.h* del código fuente. Ésta permite la interacción con los tracepoints (registro y desregistro de callbacks a un tracepoint, obtención de todos los tracepoints instalados en el sistema, etc) desde un módulo del kernel, con la inclusión de `<linux/tracepoint.h>`. La API de tracepoints es madura, ha sido bien documentada y ofrece la funcionalidad adecuada para permitir hacer que las funciones callback instaladas en tracepoints de interés puedan invocar a las callbacks de PMCTrack cuando éstas se alcancen. Esto será posible siempre que exista algún tracepoint que proporcione los argumentos necesarios para la callback de PMCTrack y se encuentre en la zona de código en la que estaba instalada la callback original por el parche de PMCTrack, o en una zona de código alternativa en la que tenga el mismo efecto.

Esta API también dispone de macros para la creación de nuevos tracepoints. Estos tracepoints nuevos pueden pasar el número arbitrario de parámetros que se deseen, sus prototipos se han de describir en una declaración de tracepoint creada mediante las macros de la API y ha de ser colocada en un fichero de cabecera, nuevo o ya existente, dentro del kernel Linux en la ruta */include/trace/events*. Los nuevos tracepoints pueden ser instalados en prácticamente cualquier lugar de interés dentro del código fuente del kernel. Todo este proceso requiere de la recompilación del kernel Linux con la inclusión de los nuevos tracepoints.

La implementación de trazado estático por tracepoints se usa también ampliamente en muchos otros frameworks y frontends de trazado como son: trace events, BPF, perf, bpftrace, Systemtap, ftrace, LTTng, …

3.4.2 Ftrace

Ftrace es el primer sistema de trazado genérico que se ha incorporado en el kernel Linux. Fue integrado en la versión 2.6.27 y ha estado en desarrollo desde 2008. Ha sido diseñado para ayudar a los desarrolladores y diseñadores de sistemas a averiguar qué sucede dentro del kernel. Puede utilizarse para depurar o analizar latencias y problemas de rendimiento que no tienen lugar en espacio de usuario [10], [45].

La infraestructura ftrace fue creada originalmente para adjuntar callbacks al principio de las funciones con el fin de registrar y trazar el flujo del kernel. No obstante, las callbacks al inicio de una función pueden tener otros casos de uso. Con ftrace se puede hacer parches en vivo en el kernel; de hecho es la tecnología base usada para la infraestructura *Kernel Live Patching* dentro del kernel Linux [53]. Ftrace permite averiguar qué funciones del kernel se invocan cuando se ejecuta una aplicación en el espacio de usuario. Se puede trazar el comportamiento de las funciones, medir el tiempo de ejecución y encontrar cuellos de botella y problemas de rendimiento. También se pueden identificar bloqueos en el espacio del kernel, medir el tiempo que tarda en ejecutarse una tarea en tiempo real, descubrir problemas de latencia, medir el uso de la pila del kernel y encontrar posibles desbordamientos de pila [45].

Ftrace ofrece, desde su integración en el kernel Linux, su propia API de trazado que permite el filtrado de funciones disponibles a trazar dentro del kernel Linux y el registro de callbacks a dichas funciones desde un módulo del kernel, disponible con la inclusión de `<linux/ftrace.h>`. Si el kernel se configura con el símbolo de preprocesado `DYNAMIC_FTRACE_WITH_REGS` las callbacks recibirán como argumento los registros del kernel con los argumentos y puntero de instrucción (ip) de la función en las que están registradas. Las callbacks son ejecutadas en el mismo contexto que lo hace la instrucción a la que se encuentran adjuntadas. Por defecto las callbacks instaladas con ftrace son ejecutadas con expropiación deshabilitada. Esto se realiza para proteger a aquellas estructuras con la información de trazado para ftrace, `ftrace_ops`, cuya memoria se asigna dinámicamente. Aun así es posible rehabilitar la expropiación dentro de la propia callback. Ftrace hace que el enganche de callbacks a las funciones del kernel Linux sea fácil y proporcione dos ventajas cruciales:

- **Su API es madura y con un código sencillo.** Aprovecha las interfaces listas para usar en el kernel, lo que reduce significativamente la complejidad del código [54]. A través de la API de ftrace se puede adjuntar callbacks a funciones del kernel desde un módulo del kernel con apenas un par de llamadas, y la definición de la estructura `ftrace_ops` con tan sólo dos campos.

- **La posibilidad de trazar cualquier función por su nombre.** El trazado del kernel Linux con ftrace es un proceso bastante simple, con la escritura del nombre de la función a trazar en una cadena regular es suficiente para indicar el punto de traza. No es necesario tratar con el enlazador, conocer direcciones de memoria de las funciones ni investigar las estructuras de datos internas del kernel. Mientras se

conozca el nombre se pueden trazar las funciones del kernel con ftrace, incluso si esas funciones no están exportadas para los módulos [54].

Ftrace presenta el inconveniente de que necesita requisitos de configuración del kernel para garantizar su correcto funcionamiento en el trazado del kernel Linux. Como ya ha sido establecido anteriormente, se necesita de al menos las siguientes configuraciones: `CONFIG_FTRACE`, `CONFIG_DYNAMIC_FTRACE_WITH_REGS` y `CONFIG_FUNCTION_TRACER` para obtener la API ftrace y realizar el trazado. Estas características pueden desactivarse en la configuración del kernel ya que no son críticas para el funcionamiento del sistema. Sin embargo, normalmente los kernels utilizados por las distribuciones populares de Linux, como son Debian, Arch, etc, mantienen todas estas opciones del kernel, ya que no afectan al rendimiento del sistema de forma significativa y pueden ser útiles para la depuración. Aun así, hay que tener en cuenta estos requisitos en caso de necesitar soporte en kernels particulares de arquitecturas específicas. Por ejemplo en ARM64 el soporte de `DYNAMIC_FTRACE_WITH_REGS` no ha sido añadido hasta la versión 5.5 del kernel Linux.

Una limitación que puede verse en ftrace es que, a diferencia de Kprobes, sólo hace trazado en el punto de entrada de una función. Puede considerarse esta limitación como una desventaja, pero para el caso concreto de este proyecto ésto no supone ninguna limitación.

La presencia en el kernel Linux de la API madura de ftrace y junto con su eficiente implementación de trazado –con el objetivo de lograr la menor sobrecarga posible– hacen de ftrace el sistema de trazado dinámico principal a usar para la adaptación de las callbacks de PMCTrack, para lograr así una implementación sin parche del kernel.

La infraestructura de ftrace además ofrece las siguientes características adicionales: soporte de tracepoints, soporte para kprobes y soporte de blktrace [55]. Ftrace también hace uso de tracefs, un pseudo sistema de archivos para configurar ftrace y recoger los datos de trazado. Todas las manipulaciones se realizan con simples operaciones sobre ficheros en el directorio */sys/kernel/debug/tracing/* del sistema. Si el trazado se activa con tracefs, todos los datos de trazado recogidos serán almacenados por ftrace en un buffer circular en memoria y expuestos en el fichero */sys/kernel/debug/tracing/trace* del sistema. Es posible incluso el acceso y uso de tracepoints con ftrace a través de tracefs en el directorio */sys/kernel/debug/tracing/trace/events* del sistema.

3.4.3 Kprobes

Kprobes (*Kernel Dynamic Probes*) es un sistema de trazado introducido en el kernel Linux desde la versión 2.6.9 en 2004. Permite irrumpir dinámicamente en cualquier rutina del kernel –sin necesidad de modificar su código fuente– y recopilar información de depuración y rendimiento de forma no disruptiva [9]. Como se ha explicado anteriormente en la sección 3.3.2.2, para lograr ésto se inserta un kprobe escribiendo dinámicamente instrucciones de interrupción en una dirección dada del kernel Linux en ejecución. La

ejecución de la instrucción trazada provoca que Kprobes tome control del manejador de la interrupción y ejecute una rutina manejadora dada para dicho kprobe [9], [50]. Actualmente existen dos tipos de kprobes:

- kprobes, se pueden insertar al comienzo de prácticamente cualquier instrucción del kernel.
- kretprobes, se ejecutan cuando una función especificada retorna.

Kprobes solo estará habilitado en el sistema si el kernel Linux es compilado con las configuraciones `CONFIG_KPROBES` y `CONFIG_KALLSYMS`. Kprobes proporciona una interfaz ligera, madura y bien documentada. Es accesible a través de `<linux/kprobes.h>` y permite desde un módulo del kernel implantar puntos de traza (kprobes) y registrar sus correspondientes manejadores. También cuenta con funciones para desactivar temporalmente un kprobe específico y para su posterior rehabilitación.

Toda esta flexibilidad tiene algunas desventajas:

- Kprobes se basa en interrupciones y manipula los registros del procesador. Así que para realizar la sincronización, todos los manejadores proporcionados a un kprobe necesitan ser ejecutados con expropiación deshabilitada. Como resultado, hay varias restricciones para los manejadores: en ellos no se puede invocar funciones bloqueantes, lo que significa que no se pueden asignar grandes cantidades de memoria, ni tratar con la entrada-salida, ni bloquearse en semáforos y funciones de temporizadores, etc [9], [56].
- Un kprobe ejecutado tiene una sobrecarga de rendimiento significativa ya que utiliza interrupciones y manejadores de excepciones. Aunque se trata de un procedimiento único, la colocación de puntos de interrupción es bastante costosa. Si bien los puntos de interrupción no afectan al resto de las funciones, su procesamiento también es relativamente costoso en rendimiento. Esta característica es el principal motivo del descarte de este sistema de trazado en la aplicación a la adaptación de las callbacks de PMCTrack a una implementación sin parche del kernel.
- Otra desventaja es la ubicación de los kprobes, se puede colocar fácilmente a la entrada y salida de una función, pero si necesita trazar dentro de una función o variables locales, entonces es preciso tener instalado SystemTap [57] y un kernel compilado con el símbolo de preprocesamiento `DEBUG_INFO` [58].

3.5 Perf_events

`Perf` es una herramienta de análisis del rendimiento orientada a eventos y está disponible en Linux 2.6.31 en adelante. Esta herramienta presenta una sencilla interfaz de línea de comandos al abstraer las diferencias del *hardware* de la CPU en las mediciones de rendimiento de Linux [2], [59], [60].

`Perf` se basa en el subsistema de eventos perf_events del kernel Linux, este subsistema

perf_events también ofrece dentro del kernel su propia API. El subsistema perf_events crea una abstracción de las fuentes de monitorización de un sistema en forma de eventos dentro del kernel. Tanto la herramienta `perf` y la API del kernel de perf_events pueden medir los eventos que provienen de diferentes fuentes [2]. El subsistema de eventos perf_events incluye eventos *hardware* (contadores *hardware* de rendimiento), eventos *software* (contadores del kernel), e incluso eventos de trazado (tracepoints, kprobes, etc). Su inclusión introdujo la posibilidad de acceder a los contadores de rendimiento con facilidad, sin necesidad de parches del kernel o recompilaciones. Este subsistema de eventos perf_events permite el acceso a los contadores de rendimiento para una amplia gama de arquitecturas y modelos de procesadores que antes no tenían soporte por defecto en el sistema operativo [5].

Aunque el subsistema perf_events no se centra sólo en el trazado dentro del código fuente del kernel, sí que ofrece funcionalidades con eventos *software* que pueden plantear su uso para lograr el objetivo de almacenar datos de monitorización por proceso para PMCTrack. En las explicaciones que prosiguen se irá entrando en detalle sobre estas características que resultan de interés sobre perf_events.

El subsistema perf_events se construye alrededor de descriptores de archivo asignados con la nueva llamada del sistema `sys_perf_event_open()`. Los eventos se especifican en el momento de su creación en una sofisticada estructura `perf_event_attr`; esta estructura tiene más de 40 campos diferentes que interactúan de forma compleja entre sí. Los eventos de perf_events se activan y desactivan mediante llamadas a `ioctl()` o `prctl()` y los valores se leen mediante la llamada del sistema estándar `read()` [5].

El principal objetivo del diseño de perf_events es proporcionar toda la funcionalidad y abstracción posible en el kernel, haciendo que la interfaz sea sencilla para los usuarios menos experimentados [5]. El código fuente de la herramienta de análisis de rendimiento `perf` y el subsistema de eventos perf_events se añadieron en 2009 al árbol principal de Linux, bajo el directorio */tools/perf* de las fuentes [2], [5]. Esta inclusión dentro del kernel es una de las ventajas de perf_events frente a otras implementaciones.

Existe una gran variedad de eventos que se pueden medir con contadores de rendimiento, la disponibilidad de eventos, no obstante, varía considerablemente entre las CPUs. Un evento puede tener sub-eventos (o máscaras de selección). En algunos procesadores y para algunos eventos, puede ser posible combinar las máscaras de selección y medir cuándo se produce cualquiera de los sub-eventos. Un evento también puede tener modificadores, es decir, filtros que alteran cuándo o cómo se cuenta el evento [5]. A continuación se exponen los tipos de eventos y sus características principales.

- **Eventos tracepoint**

  Estos eventos de trazado hacen uso de los tracepoints, explicados en puntos anteriores; perf_events implementa estos eventos mediante la infraestructura ftrace del kernel Linux y sólo están disponibles desde la versión 2.6.30 de kernel. La infraestructura perf_events permite también hacer trazado dinámico con la

anotación de eventos en el código fuente [2]. Esto es posible mediante la definición dinámica de nuevos puntos de traza a través kprobes, en espacio del kernel, y uprobes [61], en espacio de usuario.

Aunque perf_events utiliza muchas características de trazado de Linux, algunas todavía no están expuestas a través de la herramienta `perf`, y necesitan ser utilizadas en su lugar a través de la interfaz ftrace [59].

El trazado con perf_events no se puede considerar de utilidad para la adaptación de las callbacks de PMCTrack a una implementación que no requiera de un parche del kernel. Esto se debe a que hace uso de sistemas de trazado (ftrace, tracepoints, …) que ya ofrecen sus propios frameworks o APIs de trazado dentro del kernel Linux *mainline*. El uso de estos frameworks o APIs permiten un control completo sobre el trazado y no están restringidos a las funcionalidades en los casos de uso que hace perf_events.

- **Eventos *hardware***

  Este tipo de eventos provienen de contadores *hardware* de rendimiento o de unidades de monitorización del rendimiento (PMUs) presentes en cada CPU del sistema. Los eventos *hardware* varían con cada tipo y modelo de procesador y constituyen una base para perfilar las aplicaciones con el fin de rastrear el flujo de control dinámico e identificar los puntos conflictivos [2].

  Perf_events proporciona valiosas abstracciones generalizadas sobre las capacidades específicas del *hardware*:

  - Eventos generalizados: existen eventos de uso común en varias arquitecturas con nombres comunes. Éstos forman un subconjunto común de eventos útiles que están disponibles en la mayoría de las CPUs modernas como pueden ser ciclos, fallos en caché, etc. La interfaz perf_events proporciona un conjunto reducido de nombres para estos eventos *hardware* comunes. En cada procesador, estos eventos se asignan a un evento real proporcionado por la CPU si existe, de lo contrario el evento no puede ser utilizado [2], [5].
  - Soporte de nuevas características del *hardware* [5]:
    * *Eventos Offcore Response*: Los chips más recientes de Intel y los de Nehalem ofrecen eventos de respuesta fuera del *core*, que permiten medir de forma filtrada los accesos a la memoria y otras actividades que proceden del *core*.
    * *Eventos Uncore y Northbridge*: Los procesadores modernos incluyen múltiples núcleos en una sola CPU. Ésto conlleva una infraestructura compartida (cachés L2 y L3, interconexiones y controladores de memoria). Hay varios eventos que miden estos recursos compartidos.
    * *Interfaces de muestreo*: Los procesadores más recientes ofrecen interfaces de muestreo. Permiten etiquetar ciertas instrucciones y devolver al usua-

rio información más detallada. En procesadores AMD ésto se denomina IBS (*Instruction Based Sampling*) y en Intel PEBS (*Precise Event Based Sampling*).

  * *Contadores virtualizados*: El incremento de las inversiones en computación en la nube y virtualización han aumentado el interés en el uso de contadores de rendimiento dentro de las máquinas virtuales. Recientemente se ha logrado conseguir acceso a los registros *hardware* relevantes y proporcionar soporte de los contadores *hardware* para los sistemas operativos *guest*. Perf_events proporciona soporte adicional para recopilar estadísticas KVM del *host*.

- **Eventos *raw***

  Con este tipo de eventos, perf_events permite indicarle al evento la configuración que ha de usar directamente en los contadores *hardware* del rendimiento [62]. Estas configuraciones se encuentran en los manuales de los procesadores de cada fabricante. Este tipo de eventos permiten usar PMCs cuando no se encuentran abstraídos en un evento *hardware*. Esto no impide indicarle al evento *raw* configuraciones de PMCs que sí se encuentran representados en eventos *hardware*.

  La API del kernel de perf_events junto con los eventos *raw* fueron utilizados para la creación del nuevo backend perf experimental para PMCTrack, incluido en la versión 1.5 de PMCTrack. Este nuevo backend de PMCTrack consiguió, mediante el uso de la API de perf_events, soporte para procesadores Intel x86 con una monitorización de la colección de eventos reducida. Ésto expuso la posibilidad de eliminar la dependencia con la arquitecta, que presentan el resto de los backends de PMCTrack, con la ampliación de este nuevo backend experimental.

- **Eventos *software***

  Perf_events proporciona soporte para los eventos *software* del kernel que no proporciona el *hardware* [5]. Estos eventos son puros contadores del kernel como el número de los cambios de contexto o los fallos de página, y se exponen utilizando la misma interfaz que los eventos *hardware* [2]. Esto permite un acceso más fácil a través de la interfaz de perf_events.

Existen una serie de características especificas de perf_events que ofrece sobre los eventos, éstas son las siguientes:

- Planificación de eventos: Algunos eventos tienen sofisticadas restricciones de *hardware* y sólo pueden ejecutarse en un determinado subconjunto de contadores disponibles. Perf_events se encarga dentro del kernel de planificar la asignación de los eventos a los contadores adecuados [5].
- Multiplexación: Se puede desear medir más eventos simultáneamente de los que el *hardware* puede soportar físicamente. Para ello es necesario multiplexar eventos en los contadores físicos disponibles. Los eventos se ejecutan durante un breve periodo

de tiempo, luego se alternan con otros eventos y se extrapola estadísticamente un recuento total estimado. Perf_events multiplexa los eventos en el kernel utilizando un intervalo fijo de round-robin, lo que puede proporcionar un mejor rendimiento pero menos flexibilidad [5], [63].

- Conteos por proceso: perf_events proporciona contadores por tarea. Para proporcionar datos de rendimiento por proceso para eventos *hardware*, el sistema operativo guarda y restaura los registros *hardware* del contador al cambiar de contexto [2]. Dentro de perf_events también se ofrece acoplamiento a procesos, obtener contadores de rendimiento en un proceso que ya se encuentra en ejecución.

Los eventos *software* de perf_events junto con su capacidad de mantener contadores por tarea, proporcionando datos de rendimiento por proceso, plantean la posibilidad de uso de perf_events para lograr uno de los objetivos en la adaptación de PMCTrack a entornos de producción. Se trata de lograr un cambio en el modo de mantener datos de monitorización por proceso para PMCTrack dentro del sistema, de manera que no sea necesario aplicar un parche al kernel.

Capítulo 4

Patchless PMCTrack

En un principio la adaptación de la API del kernel de PMCTrack a kernels Linux vanilla se planteó llevarla a cabo dentro de un nuevo módulo del kernel, el cual debería instalarse antes de cargar el correspondiente módulo cargable de PMCTrack. Esta idea fue rápidamente reemplazada por un modelo más centralista con la inclusión de la API de PMCTrack en el módulo del kernel de PMCTrack; evitando así la carga de distintos módulos del kernel dependientes entre si. Esto implica que PMCTrack pasa a ser una herramienta cuya única instalación requerida es la carga de un sólo módulo en el kernel Linux. Este módulo puede ser o bien genérico con el backend perf o específico de la arquitectura con backends heredados de la versión anterior de PMCTrack.

Todo el código fuente del kernel Linux que se muestra a lo largo de los siguientes capítulos corresponde a la versión 5.4.35.

El resto del capítulo se estructura de la siguiente forma. En la **sección 4.1** se presenta la búsqueda de tracepoints compatibles para ser usados con las callbacks de PMCTrack. En esta sección también se explica la implementación que se ha llevado a cabo para utilizar tracepoints en las callback de PMCTrack. En la **sección 4.2** se describe la búsqueda de funciones del kernel para ser trazadas con ftrace y cómo se ha realizado el uso de ftrace para la instalación de las callbacks de PMCTrack. En la **sección 4.3** se muestran los descubrimientos realizados para poder hacer uso de perf_events en el mantenimiento de los datos de PMCTrack por tarea y la implementación realizada. En la **sección 4.4** se expone la alternativa que se ha implementado para conseguir un proceso a partir de su pid (identificador de proceso). Por último, en la **sección 4.5** se detalla la integración en las fuentes de PMCTrack donde se han incorporado las adaptaciones de la API de PMCTrack.

4.1 Uso de tracepoints para las callbacks de la API del kernel de PMCTrack

Para la aplicación de tracepoints en la adaptación de los callbacks de la API del kernel de PMCTrack se hace uso de la API de tracepoints. Esta API está presente en el código fuente del kernel Linux en */include/linux/tracepoint.h*. La API de tracepoints proporciona todas las operaciones necesarias (registro, eliminación, búsqueda, declaración, etc) para el uso de los tracepoints, asociando una callback definida por el programador a tracepoints específicos. Esta callback será ejecutada cada vez que dicho tracepoint sea alcanzado en el mismo contexto. Cuando termine su ejecución se retorna a la función invocadora, continuando desde el sitio del tracepoint. En esta sección se referirá a las callbacks instaladas por los tracepoints como función de enganche para hacer una distinción clara con las callbacks de PMCTrack.

La API de tracepoints es accesible desde un módulo del kernel incluyendo el fichero de cabecera `<linux/tracepoint.h>`. También se puede encontrar información sobre los tracepoints estáticos instalados en el sistema dentro de la ruta */sys/kernel/debug/tracing/* del sistema, donde se puede hacer uso de dichos tracepoints a través de herramientas como ftrace, perf, etc, que proporcionan un framework para su uso desde la línea de comandos.

A continuación se procede a explicar, en distintas subsecciones, la búsqueda de tracepoints que ofrecen la funcionalidad necesaria para ser usados en las callback de PMCTrack y la implementación que se ha llevado a cabo para hacer uso de dichos tracepoints dentro de PMCTrack.

4.1.1 Búsqueda de tracepoints compatibles con las callbacks de PMCTrack

La lista de todos los tracepoints estáticos disponibles se encuentra en el fichero */sys/kernel/debug/tracing/available_events* del sistema:

```
$ sudo grep sched /sys/kernel/debug/tracing/available_events
[...]
sched:sched_process_exit
sched:sched_process_free
sched:sched_migrate_task
sched:sched_switch
[...]
```

También es posible la obtención de todos los tracepoints mediante la función `for_each_kernel_tracepoint()` de la API de tracepoints del kernel. Esta función recorre todos los tracepoints instalados en el sistema. Se hará uso de ella en la parte de implementación realizada con tracepoints, que se explica más adelante en este capítulo.

Cada tracepoint estático tiene asociado un fichero de "formato" que se encuentra situado

en la ruta */sys/kernel/debug/tracing/events* del sistema y que contiene una descripción de cada campo del evento registrado. Los directorios dentro de esta ruta representan los subsistemas de trazado que están disponibles, dentro de los cuales se pueden definir múltiples tracepoints [64].

```
$ sudo ls /sys/kernel/debug/tracing/events
alarmtimer      fs_dax          mce             random          task
block           ftrace          migrate         ras             tcp
bpf_test_run    gpio            module          raw_syscalls    thermal
cgroup          header_event    mpx             rcu             timer
clk             header_page     msr             regulator       tlb
compaction      huge_memory     napi            rpm             udp
cpuhp           hwmon           neigh           rseq            vmscan
devfreq         i2c             net             rtc             vsyscall
dma_fence       initcall        nmi             sched           wbt
drm             intel_iommu     oom             scsi            workqueue
enable          iommu           page_isolation  signal          writeback
exceptions      irq             pagemap         skb             x86_fpu
ext4            irq_matrix      percpu          smbus           xdp
fib             irq_vectors     power           sock            xen
fib6            jbd2            preemptirq      spi
filelock        kmem            printk          swiotlb
filemap         libata          qdisc           syscalls
```

El formato de cada tracepoint también se puede consultar dentro del código fuente del kernel Linux. Se encuentran en el directorio *include/trace/events/*, que contiene los ficheros de cabecera para cada subsistema de trazado, en los que se declaran cada tracepoint que incluyen. El conocimiento de la declaración de los tracepoints que se desean usar, ya sea viendo su correspondiente fichero de "formato" o a través del fichero de cabecera en el código fuente del kernel Linux, es importante ya que se han de conocer sus argumentos y el formato en el que se ofrecen.

Para poder localizar los tracepoints en el código fuente del kernel Linux, se ha de tener en cuenta que la convención es que los tracepoints se llaman a través del siguiente formato de función: *trace_<tracepoint_name>*. A continuación se analizan los resultados de la búsqueda de tracepoints en las zonas del código fuente en las que el parche de PMCTrack instala las callbacks.

En el caso de la callback **pmcs_save_callback**, se encuentra instalada en el parche de PMCTrack en la función `__schedule()` del kernel. Si se consulta esta función en el código fuente del kernel Linux (en el fichero *kernel/sched/core.c*) se comprueba que para esta callback existe el tracepoint **sched_switch** dentro de esa función:

```
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    [...]
    int cpu;
```

```
    cpu = smp_processor_id();
    [...]
        ++*switch_count;

        trace_sched_switch(preempt, prev, next);

        /* Also unlocks the rq: */
        rq = context_switch(rq, prev, next, &rf);
    [...]
}
```

La llamada **pmcs_save_callback** acepta dos argumentos: el `task_struct` de la tarea previa (la que "sale" de la CPU en el cambio de contexto) y el número de la CPU actual. Este tracepoint, **sched_switch**, proporciona como argumento el `task_struct` de la tarea previa y, aunque no proporciona el número de la CPU actual, no resulta un problema puesto que este dato se consigue haciendo convenientemente la llamada a la función `smp_processor_id()`. Esto permite que se pueda obtener fácilmente dentro de la propia función de enganche asignada al tracepoint, ya que ésta se ejecuta en el mismo contexto que `__schedule()`. Establecido todo lo comentado, el tracepoint **sched_switch** ofrece las características adecuadas para poder adaptar a **pmcs_save_callback** fuera del código fuente del kernel.

En cuanto a la callback **pmcs_exit_thread**, se instala mediante el parche del kernel de PMCTrack en la función `do_exit()` del código fuente de Linux. De la misma manera (consultando en el fichero *kernel/exit.c* del kernel) se encuentra que para esta callback existe el tracepoint estático **sched_process_exit** dentro de la función `do_exit()` del kernel:

```
void __noreturn do_exit(long code)
{
    struct task_struct *tsk = current;
    int group_dead;
    [...]
    if (group_dead)
        acct_process();
    trace_sched_process_exit(tsk);

    exit_sem(tsk);
    [...]
}
```

En la llamada **pmcs_exit_thread** sólo se requiere pasar un argumento, el `task_struct` de la tarea actual. Convenientemente el tracepoint **sched_process_exit** proporciona precisamente este dato como su único argumento. Esto hace que este tracepoint sea ideal para sustituir a **pmcs_exit_thread**.

En el caso de la callback **pmcs_free_per_thread_data**, se invoca cuando se libera el `task_struct` de una tarea del sistema. En la función `free_task()` (del fichero

kernel/fork.c de las fuentes de Linux) donde se encuentra instalada, no existe ningún tracepoint. Aun así existe el tracepoint **sched_process_free**, pero éste se encuentra instalado en la función `delayed_put_task_struct()` en el fichero *kernel/exit.c* que, aunque se podría haber considerado candidato a ser utilizado como callback, tras ser investigado su uso se determinó su incompatibilidad como callback para PMCTrack.

Algo similar sucede con el callback **pmcs_exec_thread** en la función `sched_exec()` del fichero *kernel/sched/core.c* de las fuentes de Linux. No hay instalado ningún tracepoint en dicha función pero sí existe el tracepoint **sched_process_exec** situado en el fichero *fs/exec.c* en la función `exec_binprm()` del kernel. Este tracepoint tampoco es compatible con la callback de PMCTrack.

Para la callback **pmcs_alloc_per_thread_data**, situada en la función `sched_fork()` (del fichero *kernel/sched/core.c* de las fuentes de Linux), tampoco existe ningún tracepoint en dicha función. Dentro de los tracepoints disponibles, se encuentra uno en particular que podría solucionar esta callback que es **sched_process_fork**, pero supone dos compromisos usarlo. El primero y más grave, es el alejamiento de la callback del momento ideal en que ésta debe ser recibida por PMCTrack. El retardo que supondría recibir la callback (en la creación de un nuevo hilo) tras la realización de un *fork* por el sistema y no el instante que sucede, puede causar un comportamiento inestable en PMCTrack. El segundo compromiso viene de los argumentos que ofrece este tracepoint; los argumentos no incluyen el dato referente a los `clone_flags` (propiedades de duplicación que se han de tomar al hacer el fork de un proceso [65]). Aunque este inconveniente sería solventable su solución tampoco se acercaría a un resultado cercano al ideal esperado.

En el resto de callbacks instaladas por el parche de PMCTrack, **pmcs_restore_callback** y **pmcs_tbs_tick**, no existen tracepoints que ayuden a sustituirlas. Debido a esto y a que los tracepoints encontrados como alternativa no son factibles para las callbacks **pmcs_free_per_thread_data**, **pmcs_alloc_per_thread_data** y **pmcs_exec_thread**, los tracepoints no proporcionan una solución completa para la adaptación de las callbacks de PMCTrack. Esto obliga a emplear otras tecnologías de trazado que permitan cumplir la adaptación del resto de callbacks.

4.1.2 Uso de tracepoints del kernel Linux

Para la utilización de tracepoints se ha de conocer primero la descripción del prototipo de la función que se pretende adjuntar a los tracepoints de interés: **sched_switch** y **sched_process_exit**. A continuación se procede a exponer el formato de dichos tracepoints, según se indica en su fichero cabecera *include/trace/events/sched.h* del kernel, y la creación de sus funciones de enganche correspondientes.

El tracepoint **sched_switch** tiene el siguiente formato definido en el kernel:

```
/*
 * Tracepoint for task switches, performed by the scheduler:
 */
```

```
TRACE_EVENT(sched_switch,

    TP_PROTO(bool preempt,
         struct task_struct *prev,
         struct task_struct *next),

    TP_ARGS(preempt, prev, next),
    [...]
);
```

De este modo la función de enganche se ha de codificar con la siguiente declaración:

```
static void probe_sched_switch(void *data, bool preempt, struct task_struct *prev,
                               struct task_struct *next)
{
    pmcs_save_callback(prev, smp_processor_id());
}
```

Dentro de esta función de enganche se llama al callback de PMCTrack **pmcs_save_callback** con el `task_struct` de la tarea previa, argumento que se obtiene del tracepoint, y el número de la CPU actual que –al igual que se hace en la función `__schedule`– se puede conseguir simplemente haciendo una llamada a la función `smp_processor_id()`.

Para el tracepoint **sched_process_exit** el formato –definido en el código fuente del kernel– que le corresponde es el siguiente:

```
/*
 * Tracepoint for a task exiting:
 */
DEFINE_EVENT(sched_process_template, sched_process_exit,
         TP_PROTO(struct task_struct *p),
         TP_ARGS(p));
```

Esto implica que la creación de su función de enganche en PMCTrack consiste en la siguiente implementación:

```
static void probe_sched_process_exit(void *data, struct task_struct *p)
{
    preempt_enable_notrace();
    pmcs_exit_thread(p);
    preempt_disable_notrace();
}
```

Dentro de esta función de enganche se llama al callback de PMCTrack **pmcs_exit_thread** con el `task_struct` de la tarea actual, argumento heredado del tracepoint. El motivo por el cual se rehabilita la expropiación dentro de esta función de enganche se explica en detalle en el capítulo 5.

Para realizar el registro de las funciones de enganche que se acaban de describir es necesario buscar los tracepoints que se pretenden usar entre todos lo que existen en el kernel y conseguir su información asociada. Dicha información está recogida en el

tipo de datos `struct tracepoint` (definido en *include/linux/tracepoint-defs.h* de las fuentes de Linux), que permite registrar funciones de enganche a un tracepoint dado. Para guardar estos datos por cada tracepoint de interés se ha creado una estructura de datos en PMCTrack a la que se ha llamado `tracepoint_table` con los siguientes campos:

```
struct tracepoint_table {
    const char *name;
    void *fct;
    struct tracepoint *value;
    char init;
};
```

- **name**, es el nombre del tracepoint al que corresponde.
- **fct**, es la función de enganche que se proporciona para ser ejecutada cada vez que se alcance el tracepoint.
- **value**, es la información asignada a cada tracepoint en su creación y que permite la asignación de una función de enganche a dicho tracepoint.
- **init**, se trata de un flag que comprueba si ha sido registrada una función de enganche al tracepoint, lo que se asemeja a decir que el tracepoint esta "activado".

Estos datos correspondientes a cada tracepoint de interés, **sched_switch** y **sched_process_exit**, son guardados en PMCTrack en un array con una entrada para cada uno, facilitando la inclusión de nuevos tracepoints en el futuro:

```
struct tracepoint_table interests[] = {
    {.name = "sched_switch", .fct = probe_sched_switch},
    {.name = "sched_process_exit", .fct = probe_sched_process_exit},
};
```

Afortunadamente, aunque la operación de búsqueda de tracepoints puede ser costosa debido a la gran cantidad de tracepoints instalados en el kernel Linux, sólo se ha de realizar una vez: en el momento que se carga el módulo en cuestión en el kernel Linux. La búsqueda de tracepoints se realiza mediante la función `for_each_kernel_tracepoint()`, proporcionada por la API de tracepoints del kernel Linux, que itera por todos los tracepoints del kernel. En el caso de PMCTrack interesa conseguir los datos asociados (`struct tracepoint`) a los tracepoints **sched_switch** y **sched_process_exit** para posteriormente registrar las funciones de enganche asociadas, discutidas anteriormente. Para lograr esto hay que tener en cuenta que la función `for_each_kernel_tracepoint()` acepta como uno de sus argumentos una función callback con el siguiente prototipo `void (*fct)(struct tracepoint *tp, void *priv)`, la cual será ejecutada para cada tracepoint instalado en el kernel. Esta función recibe como argumento precisamente el `struct tracepoint` asociado al tracepoint en el que está iterando. Es por esto que se ha creado la siguiente función, `lookup_tracepoints()`, en el módulo de PMCTrack para ser establecida como argumento de `for_each_kernel_tracepoint()`:

```
static void lookup_tracepoints(struct tracepoint *tp, void *ignore)
{
```

```
    int i;
    FOR_EACH_INTEREST(i) {
        if (strcmp(interests[i].name, tp->name) == 0) interests[i].value = tp;
    }
}
```

En esta nueva función se compara el nombre del tracepoint iterado con **sched_switch** y **sched_process_exit**. En caso de corresponderse a alguno de los dos, se guarda su `struct tracepoint` en el campo `value` de su estructura `tracepoint_table`.

Una vez conseguido el `struct tracepoint` de **sched_switch** y **sched_process_exit** ya se está en disposición de asignarle a cada tracepoint su función de enganche, lo que se puede traducir en "activar" el tracepoint. Este registro se hace mediante la función `tracepoint_probe_register()` –proporcionada también por la API de tracepoints del kernel Linux– que conecta una función de enganche a un tracepoint indicado. Esta función, `tracepoint_probe_register()`, ha de ser llamada para cada tracepoint de interés con su correspondiente puntero a `struct tracepoint` y puntero a la función de enganche que se le quiere asignar. Hay que tener en cuenta que el trazado de estos tracepoints sólo ha de realizarse cuando el módulo de PMCTrack esté instalado en el kernel Linux. Cuando módulo de PMCTrack quiera ser descargado del kernel Linux es necesario eliminar la asignación de las funciones de enganche de PMCTrack, "desactivar" los tracepoints. Esto se realiza mediante la función proporcionada por la API de tracepoints del kernel `tracepoint_probe_unregister()`, la cual desconecta una función de enganche de un tracepoint. En el módulo de PMCTrack su llamada vendrá dictada por el flag `init` (de la estructura `tracepoint_table` de cada tracepoint), que sólo permitirá la eliminación del registro del tracepoint asociado cuando éste se encuentre activo.

4.2 Uso de ftrace para las callbacks de la API de PMCTrack

Como ya se ha expuesto en la sección 3.4.2, la infraestructura ftrace se creó originalmente para conectar callbacks al inicio de las funciones con el fin de registrar y trazar el flujo del kernel. A partir de este momento, se hará referencia a estas callbacks como funciones de enganche, para poder hacer una distinción clara entre las callbacks de ftrace y las callbacks de la API del kernel de PMCTrack. Aunque estas funciones de enganche al inicio de una función pueden tener otros casos de uso, como la aplicación de parches en vivo en el kernel [11] o para la supervisión de seguridad [66], en el caso de sustituir las callbacks de la API del kernel de PMCTrack el uso que se les da es simplemente su registro en funciones de interés dentro del kernel Linux. Esto se realiza para poder hacer la llamada a las callbacks de PMCTrack correspondientes cada vez que estas funciones de interés sean invocadas. En el capítulo 5 se verá con detalle cómo hacer uso de ftrace para realizar un parche en vivo en el kernel Linux.

Para hacer uso de la API de ftrace dentro de un módulo del kernel se ha de incluir el fichero de cabecera `<linux/ftrace.h>`. A continuación se explica cómo se ha realizado la utilización de la API de ftrace para implementar funciones de enganche propias y conectarlas a las funciones de interés de manera que sustituyan a las callbacks del parche de la API del kernel de PMCTrack.

4.2.1 Búsqueda de funciones compatibles con las callbacks de PMCTrack

El primer paso para sustituir las callbacks de la API del kernel de PMCTrack, es establecer qué funciones del código fuente del kernel Linux son de interés. La elección de dichas funciones, aunque puede resultar directa a primera vista, se puede ver afectada por dos aspectos principales:

- Los argumentos que recibe la función escogida serán los mismos que los argumentos a los que se tendrá acceso en la función de enganche con el callback de PMCTrack. Por tanto, es necesario o bien compatibilidad de argumentos entre ambas funciones o alternativas para conseguir los mismos datos que se pasan al callback de PMCTrack.
- La capacidad de poder ser filtradas por ftrace por su nombre.

Para conocer si se puede usar el nombre de dichas funciones o por el contrario es necesario averiguar su punto de traza, que debe ser la dirección donde se encuentra la llamada a `fentry` o `mcount` en la función (véase sección 3.3.2.1), se ha de consultar el fichero */sys/kernel/debug/tracing/available_filter_functions* del sistema.

Analizando el código fuente del kernel Linux –donde se instalan el resto de callbacks en el parche de PMCTrack que no pueden ser sustituidas por traza de tracepoints– se obtienen los siguientes resultados:

- La callback **pmcs_free_per_thread_data** toma como argumentos el `task_struct` de la tarea que se le pasa a la función en la que se instala. Esta función es `free_task()` que tiene como argumento el `task_struct` de interés para el callback. Además comprobando en el fichero *available_filter_functions* es posible hacer filtro por medio del nombre con ftrace :

```
$ sudo grep free_task /sys/kernel/debug/tracing/available_filter_functions
free_task
__delayed_free_task
```

- La callback **pmcs_tbs_tick** tiene como argumentos los datos de monitorización de PMCTrack, `pmc`, obtenidos del `task_struct` de la tarea actual y el número de la CPU actual. Aunque la función en la que está instalada, `scheduler_tick()`, no tiene argumentos, como la función de enganche que se instala se ejecuta en el mismo contexto, ambos datos pueden conseguirse de manera sencilla. El `task_struct` de la tarea actual se puede obtener mediante la macro `current`

–puntero al proceso que se está ejecutando actualmente– del cual se puede obtener los datos de monitorización de PMCTrack. Esta macro se encuentra definida en el fichero de cabecera *<linux/current.h>* del sistema. El número de la CPU actual se obtiene con la función `smp_processor_id()`, definida en *<linux/smp.h>*. También es posible filtrar `scheduler_tick()` por su nombre con ftrace.

```
$ sudo grep scheduler_tick /sys/kernel/debug/tracing/available_filter_functions
scheduler_tick
```

- La callback **pmcs_exec_thread** acepta como único argumento el `task_struct` del proceso actual en ejecución. Al igual que la callback anterior, aunque este dato no se proporciona en la función que está instalada esta callback de PMCTrack, `sched_exec()`, se puede obtener mediante la macro `current`. Esta función puede ser filtrada con ftrace por su nombre:

```
$ sudo grep sched_exec /sys/kernel/debug/tracing/available_filter_functions
sched_exec
```

- La callback **pmcs_alloc_per_thread_data** recibe como argumentos el `task_struct`, del nuevo proceso resultado de hacer el *fork*, y sus `clone_flags`, propiedades de duplicación que se han de tomar al hacer el *fork* del proceso. Estos dos argumentos son los mismos que los de la función `sched_fork()` en la cual está instalada la callback y es posible filtrarla por su nombre con ftrace. Aunque en un principio se usó la función `sched_fork()` para instalar la función de enganche con el callback de PMCTrack, posteriormente fue sustituida por el uso de la función `copy_semundo()`, situada en *kernel/fork.c* del sistema. Los motivos de dicho cambio no son relevantes en este punto y se explican posteriormente en la sección 4.3.1. En el caso de esta función, `copy_semundo()`, sus argumentos también coinciden con los de la callback **pmcs_alloc_per_thread_data** y se puede filtrar por su nombre con ftrace.

```
$ sudo grep sched_fork /sys/kernel/debug/tracing/available_filter_functions
__sched_fork
sched_fork
$ sudo grep copy_semundo /sys/kernel/debug/tracing/available_filter_functions
copy_semundo
```

- La callback **pmcs_restore_callback**, necesita como argumentos los datos de *profiling* de PMCTrack dentro del `task_struct` del proceso actual y el número de la CPU actual. El único argumento de la función en la que está instalada la callback, `finish_task_switch()`, es el `task_struct` del hilo del que se acaba de cambiar (`prev`). Aunque en ese momento considerar que `prev` es igual a `current` es correcto, `prev` puede haberse movido a otra CPU. Esto hace que este callback necesite `current` en vez de `prev`. Como se ha visto anteriormente, ambos argumentos se pueden obtener dentro de la función de enganche que le sea asignada a `finish_task_switch()` y también es posible filtrarla por su nombre con ftrace.

```
$ sudo grep finish_task_switch /sys/kernel/debug/tracing/available_filter_functions
finish_task_switch
```

Desafortunadamente el trazado sobre `finish_task_switch()`, para la callback **pmcs_restore_callback**, no es posible en versiones Linux inferiores a la 5.9.7 debido a un *bug* conocido [67]. Este error consiste en que cualquier callback asociada por ftrace a `finish_task_switch()` nunca llega a ejecutarse. Para entender el por qué de este fallo en el trazado de esta función es necesario explicar de forma general la expropiación dentro de Linux.

En un sistema multitarea como Linux, ningún hilo de ejecución tiene garantizado el acceso exclusivo al procesador. El núcleo puede (casi) siempre expropiar a un hilo en ejecución en favor de otro que tenga mayor prioridad. Ese nuevo hilo puede ser un proceso diferente, pero también puede ser una interrupción de *hardware* u otro evento externo. Para coordinar adecuadamente la ejecución de todas las tareas de un sistema, el kernel debe mantener un registro del estado de ejecución actual, incluyendo cualquier tarea que haya sido expropiada o que pueda impedir que un hilo sea expropiado [68].

Una pieza de la infraestructura para ese seguimiento es el contador de expropiación que se almacena con cada tarea en el sistema. A ese contador se accede a través de la función `preempt_count()`. El propósito de este contador es describir el estado actual de cualquier hilo que se esté ejecutando, si puede ser expropiado y si se le permite dormir. Para ello debe hacer un seguimiento de una serie de estados diferentes, por lo que el contador de expropiación se divide en varios subcampos [68]. El código del kernel utiliza `preempt_count()` para tomar decisiones sobre qué acciones son posibles en un momento dado, aún así se ha de tener en cuenta que existe un retraso desde que cambia el contexto hasta que `preempt_count()` muestra el valor actualizado.

En el caso de las callbacks adjuntadas por ftrace, éste intenta protegerlas de producir recursión para impedir que el sistema entre en un bucle infinito de llamadas causando el bloqueo del sistema. Para lograr esto se utiliza `preempt_count()` para detectar la recursión haciendo que cada contexto tenga su propio bit que se establece cuando se inicia el trazado. Antes de ejecutar la callback ftrace recupera este bit con `trace_get_context_bit()` y, si está activado, se considera que está sucediendo recursión y la callback no se ejecuta [69]. Por el contrario, si esto ocurre cuando el contexto ha cambiado pero `preempt_count()` no se ha actualizado, esto se interpretará incorrectamente como una recursión, produciendo que las callback acopladas no se ejecuten, como sucede al trazar la función `finish_task_switch()`.

La solución a este problema se propuso en la versión 5.4 de Linux [69] pero no ha sido hasta la versión 5.9.7 que se ha introducido en el Linux *mainline* [67], [70]. Para solventar este error, se ha creado un nuevo bit dentro de ftrace llamado `TRACE_TRANSITION_BIT` que comprueba si el bit del contexto actual ya está establecido. La llamada se interpretará como una recursión si el bit `TRACE_TRANSITION_BIT` ya está establecido, y si no, se establece y se continúa con la ejecución de la callback proporcionada a ftrace aunque el bit

del contexto actual esté activo. El bit `TRACE_TRANSITION_BIT` se borrará normalmente al retornar la callback que lo estableció. En el caso de que el bit de contexto actual se restablezca, entonces se borra el bit `TRACE_TRANSITION_BIT` para permitir otra transición entre el contexto actual y uno aún mayor [69].

No disponer de esta callback para versiones inferiores a la 5.9.7 de Linux implica que el ofrecimiento de las métricas de monitorización a componentes del kernel no esté disponible en PMCTrack. Sin embargo ésto no afecta a la funcionalidad de monitorización de aplicaciones con PMCTrack desde espacio de usuario.

El uso de otra función dentro del kernel para trazar para la callback **pmcs_restore_callback** con ftrace no es posible ya que, por desgracia, el resto de funciones que ofrecerían un resultado similar no tienen compatibilidad con ftrace para ser trazadas.

Por otro lado el uso de kprobes como alternativa a instalar esta callback en versiones de Linux inferiores a la 5.9.7 no es viable para PMCTrack. La frecuencia de invocación de la función `finish_task_switch()` es extremadamente alta, lo que causaría una sobrecarga muy considerable en el sistema afectando negativamente a la monitorización de PMCTrack produciendo métricas peores de lo que se esperaría. En caso de querer esta funcionalidad en versiones de Linux inferiores a la 5.9.7, se ha de hacer uso de los parches del kernel de PMCTrack.

4.2.2 Funciones de enganche para PMCTrack

A partir de la versión 4.14 de Linux el prototipo de la función de enganche para ftrace es el siguiente [66]:

```
void callback_func(unsigned long ip, unsigned long parent_ip,
                   struct ftrace_ops *op, struct pt_regs *regs);
```

- **ip**, es la dirección de memoria de la función que se está trazando. Donde se encuentra `fentry` o `mcount` en la función.
- **parent_ip**, es la dirección de memoria de la función que llamó a la función que se está trazando, donde se produjo la llamada de la función trazada.
- **op**, es el puntero a la estructura `ftrace_ops` que se utilizó para registrar la función de enganche. Se puede utilizar para pasar datos a la función de enganche a través del puntero de su campo `private`.
- **regs**, dependiendo de si uno de los flags `FTRACE_OPS_FL_SAVE_REGS` o `FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED` está establecido en la estructura `ftrace_ops` registrada, entonces este argumento apuntará a la estructura `pt_regs` como lo haría si un punto de interrupción fuera colocado al comienzo de la función donde ftrace estaba trazando. De lo contrario contiene basura o `NULL`. Tanto la estructura `ftrace_ops` como sus flags son explicados en la siguiente sección. A través de este argumento se permite el acceso a los argumentos de la función que se está trazando con ftrace. Este acceso implica una dependencia con la arquitectura del sistema que a continuación se explica como se ha solventado.

La estructura `pt_regs` define la forma en la que se almacenan los registros en la pila del kernel durante una llamada al sistema u otra entrada del kernel. Se ha de conocer que existe una convención de llamada que rige cómo se pasan los argumentos de una función y los valores de retorno. Por ejemplo, en Linux x86-64, los primeros seis argumentos de función se pasan en los registros %rdi, %rsi, %rdx, %rcx, %r8 y %r9, respectivamente. El séptimo argumento y los siguientes se pasan a la pila y el valor de retorno se pasa en el registro %rax [71]. Esta convención varía según la arquitectura, lo que plantea en un principio una implementación distinta al llamar al callback de PMCTrack por cada arquitectura soportada. Afortunadamente desde la versión 4.20.0 del kernel para x86 y la versión 5.2.0 para arm64, se ha incorporado la función `regs_get_kernel_argument()`. Se trata de una función *wrapper*, que obtiene el puntero al argumento –en la posición indicada– de una función en el kernel, según la arquitectura del sistema. Para otorgar compatibilidad con versiones anteriores se ha incluido en PMCTrack la misma implementación de esta función para cada arquitectura correspondiente (*backport*).

De esta manera, a continuación se muestra como ejemplo la implementación de la función enganche para la callback **pmcs_alloc_per_thread_data** que se ha creado en PMCTrack. En ésta se pone en uso el acceso a los registros en la pila del kernel.

```
static void notrace fh_ftrace_sched_fork(unsigned long ip, unsigned long parent_ip,
                                  struct ftrace_ops *ops, struct pt_regs *regs)
{
    struct task_struct* p=(struct task_struct*) regs_get_kernel_argument(regs, 1);
    preempt_enable_notrace();
    pmcs_alloc_per_thread_data(regs_get_kernel_argument(regs, 0), p);
    preempt_disable_notrace();
}
```

La justificación del uso de las funciones de habilitación y deshabilitación de expropiación, `preempt_enable_notrace()` y `preempt_disable_notrace()`, en esta callback se expondrá en el siguiente capítulo.

4.2.3 Establecimiento de la estructura `ftrace_ops`

Para registrar una función de enganche a una función se requiere una estructura `ftrace_ops`, definida en el código fuente del kernel Linux:

```
struct ftrace_ops {
    ftrace_func_t               func;
    struct ftrace_ops __rcu     *next;
    unsigned long               flags;
    void                        *private;
    ftrace_func_t               saved_func;
#ifdef CONFIG_DYNAMIC_FTRACE
    struct ftrace_ops_hash      local_hash;
    struct ftrace_ops_hash      *func_hash;
    struct ftrace_ops_hash      old_hash;
```

```
    unsigned long               trampoline;
    unsigned long               trampoline_size;
#endif
};
```

Esta estructura se utiliza para indicar a ftrace qué función debe emplearse como función de enganche, campo `func`, así como qué protecciones realizará la función de enganche y no requerirá que ftrace las maneje.

Para guardar cada estructura `ftrace_ops` asociada a cada función de enganche y el nombre de la función donde será instalada, se ha creado la estructura `ftrace_hook` en PMCTrack:

```
struct ftrace_hook {
    unsigned char *name;
    struct ftrace_ops ops;
};
```

Empleando esta estructura se crea en PMCTrack un array, `ftrace_hooks`, con una entrada por cada función de interés dentro del kernel a la que se pretende registrar una función de enganche. En `ftrace_hooks` posteriormente será completada cada entrada con su `ftrace_ops` correspondiente.

```
static struct ftrace_hook ftrace_hooks[] = {
    {.name = "free_task"},
    {.name = "copy_semundo"},   // sched_fork
    {.name = "sched_exec"},
    {.name = "scheduler_tick"},
    {.name = "finish_task_switch"},
    {.name = "do_exit"},
};
```

Al registrar un `ftrace_ops` con ftrace, aunque dentro de la estructura `ftrace_ops` sólo el campo `func` es necesario de establecer siendo el resto opcionales, es de especial interés conocer el campo `flags`. Estos flags están definidos y documentados en *include/linux/ftrace.h* del kernel. Algunos se utilizan para la infraestructura interna de ftrace, pero otros son de gran importancia conocer sus propiedades [66], por lo que se describen a continuación:

- **FTRACE_OPS_FL_SAVE_REGS**

  Si la función de enganche requiere leer o modificar los registros que le son pasados en la estructura `pt_regs`, entonces se debe establecer este flag. El registro de un `ftrace_ops` con este flag establecido en una arquitectura que no soporta el paso de `pt_regs` a la función de enganche fallará. Para el caso concreto de las callbacks **pmcs_alloc_per_thread_data** y **pmcs_free_per_thread_data** de PMCTrack, este flag es necesario ya que a través del `pt_regs` pueden acceder a los argumentos que se pasan a través de la función de enganche. Si este flag no se activa entonces `pt_regs` o bien contiene basura o bien es `NULL`, por lo que el

acceso a los argumentos de la función de enganche no es posible.

Salvar el contexto (registros) en la pila del kernel requiere tiempo de procesamiento que puede causar el aumento de la sobrecarga de la función de enganche si se realiza en aquellas que no lo requieran. Por ello este flag se debe indicar sólo para las callbacks **pmcs_free_per_thread_data** y **pmcs_alloc_per_thread_data**, ya que para el resto el acceso a los datos no se realiza a través de los argumentos que ofrece la función de enganche.

- **FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED**

  Este flag es similar a `FTRACE_OPS_FL_SAVE_REGS`, pero el registro de un `ftrace_ops` en una arquitectura que no soporta el paso de `pt_regs` no fallará con este flag establecido. La función de enganche debe comprobar si `pt_regs` es `NULL` o basura o ninguna de las dos para determinar si la arquitectura lo soporta. Por este motivo, y ya que PMCTrack sólo puede funcionar si sus callbacks reciben los argumentos de los que requieren, no se hace uso de este flag sino de `FTRACE_OPS_FL_SAVE_REGS` y se exige que el sistema que tenga soporte para ello. Este soporte requiere que el kernel en ejecución haya sido configurado con la opción `CONFIG_DYNAMIC_FTRACE_WITH_REGS`.

- **FTRACE_OPS_FL_RCU**

  Si se establece este flag, entonces la función de enganche sólo será invocada por funciones en las que esté activo el mecanismo RCU[1] [26], [72]. Este flag también es necesario si en la función de enganche se realiza cualquier operación `rcu_read_lock()`, este es el caso de las funciones de enganche para los callbacks **pmcs_restore_callback**, **pmcs_exec_thread** y **pmcs_tbs_tick**.

  También se ha de tener en cuenta que RCU deja de vigilar en las siguientes situaciones: el sistema entra en reposo, el tiempo en el que una CPU es inhabilitada y vuelve a estar en funcionamiento, y al entrar desde el kernel al espacio de usuario y de vuelta al espacio del kernel. Durante estas transiciones, una función de enganche puede ser ejecutada y la sincronización de RCU no la protegerá.

- **FTRACE_OPS_FL_IPMODIFY**

  Si la función de enganche es para reemplazar por completo la función trazada es decir, llamar a otra función en lugar de la función trazada tras la ejecución de la función de enganche, se requiere establecer este flag. Ésta es la funcionalidad en la que se basan los parches del kernel en vivo [11]. Sin este flag el campo `ip`, puntero de retorno a la instrucción de la función que se está trazando, de la estructura `pt_regs` no puede ser modificado. Este flag requiere que se establezca también el flag

[1] RCU (*Read-Copy Update*) es un mecanismo de sincronización optimizado para situaciones de lectura. La idea básica de la RCU es dividir las operaciones destructivas en dos partes, una que impide que nadie vea el elemento de datos que se está destruyendo, y otra que realmente lleva a cabo la destrucción.

`FTRACE_OPS_FL_SAVE_REGS`. Sólo un `ftrace_ops` con `FTRACE_OPS_FL_IPMODIFY` establecido puede ser registrado a la vez en cualquier función.

Este flag, aunque no es de interés para la tarea de sustituir las callbacks de PMCTrack, será útil para conseguir la compatibilidad entre los backends distintos al de perf_events –heredados de la versión de PMCTrack con parche del kernel– y la nueva implementación de PMCTrack sin parche del kernel Linux. La creación de esta compatibilidad se expone en la sección 5.5.

Los flags `FTRACE_OPS_FL_SAVE_REGS`, `FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED`, `FTRACE_OPS_FL_STUB` y `FTRACE_OPS_FL_IPMODIFY` son un tipo de flags de atributos que sólo pueden establecerse antes de registrar las `ftrace_ops`, y no se pueden modificar mientras están registradas las funciones de enganche. Si se cambian estos atributos después de registrar `ftrace_ops`, pueden producirse resultados inesperados [66].

Una vez conocidos los flags de `ftrace_ops`, es necesario establecer el flag `FTRACE_OPS_FL_RCU` en las callbacks **pmcs_restore_callback**, **pmcs_exec_thread** y **pmcs_tbs_tick** y `FTRACE_OPS_FL_SAVE_REGS` sólo para las callbacks **pmcs_alloc_per_thread_data** y **pmcs_free_per_thread_data**.

4.2.4 Filtrado de las funciones de interés a trazar con ftrace

Una vez definidas todas las funciones de enganche para cada función a trazar y sus estructuras `ftrace_ops` correspondientes, es necesario establecer desde qué función ha de ser llamada cada función de enganche. En el caso de las funciones de enganche para los callback de PMCTrack, cada una sólo debe invocarse desde una única función específica, por lo que se debe establecer un filtro en ftrace. Los filtros se añaden por nombre, o dirección de memoria del punto de traza si se conoce. Como se ha visto anteriormente en la sección 4.2.1, afortunadamente para todas las funciones de interés –para las callback de PMCTrack– el filtro con ftrace se puede hacer por medio del nombre. Dicha manera de establecer el filtro se realiza mediante la siguiente función ofrecida por la API de ftrace en el kernel Linux:

```
int ftrace_set_filter(struct ftrace_ops *ops, unsigned char *buf,
                      int len, int reset);
```

Los parámetros de esta función tienen los siguientes propositos:

- **ops**, es la estructura `ftrace_ops` que contiene la función para hacer *profiling* y con la que se establece el filtro.
- **buf**, es la cadena que contiene el nombre de la función a filtrar.
- **len**, es la longitud de la cadena.
- **reset**, es el valor que indica si se han de restablecer todos los filtros antes de aplicar este filtro, en tal caso debe ser distinto a cero.

La función `ftrace_set_filter()` establece una función del kernel para filtrar en ftrace, denotando en una lista "*filter*" qué funciones deben ser habilitadas cuando se activa el

trazado. Dentro de PMCTrack se añade el filtro para una única función de interés de la siguiente manera:

```
ftrace_set_filter(&hook->ops, hook->name, strlen(hook->name), 0);
```

De la misma forma, pueden añadirse a una lista "*notrace*" que impedirá que esas funciones llamen a la función de enganche. Esto se realiza mediante la función `ftrace_set_notrace()` de la API de ftrace. La lista "*notrace*" esta separada de la lista "*filter*" y tiene prioridad sobre ésta. Si las dos listas no están vacías y contienen las mismas funciones, la función de enganche no será llamada por ninguna función.

```
int ftrace_set_notrace(struct ftrace_ops *ops, unsigned char *buf,
            int len, int reset);
```

Esta función toma los mismos parámetros que `ftrace_set_filter()` pero añadirá las funciones que encuentre para no ser trazadas. Un valor de `reset` distinto de cero borrará la lista "*notrace*" antes de añadir funciones –que coincidan con el argumento `buf`– a dicha lista. De la siguiente manera se borra el filtro para una única función en PMCTrack:

```
ftrace_set_notrace(&hook->ops, hook->name, strlen(hook->name), 0);
```

4.2.5 Activar el trazado con ftrace

Establecido el filtro para una función, ya se puede activar ftrace para empezar a trazar dicha función. Para activar la llamada de trazado en PMCTrack es preciso invocar la siguiente función de la API de ftrace:

```
register_ftrace_function(&hook->ops);
```

La función de enganche registrada empezará a ser llamada algún tiempo después de que se llame a `register_ftrace_function()` y antes de que ésta retorne. El momento exacto en que las funciones de enganche comienzan a ser llamadas depende de la arquitectura y la programación de los servicios. La propia función de enganche tendrá que manejar cualquier sincronización si debe comenzar en un momento exacto [66].

Para deshabilitar el trazado ha de realizarse la siguiente llamada a la función de la API de ftrace dentro de PMCTrack:

```
unregister_ftrace_function(&hook->ops);
```

La función `unregister_ftrace_function()` garantizará que la función de enganche ya no sea llamada después de su retorno [66].

4.3 Eliminación de los campos de PMCTrack en el `task_struct` con perf_events

Como se explica en la sección 2.3.2 PMCTrack precisa disponer de datos de monitorización en cada proceso del sistema, para lo cual incluye dos nuevos campos en el `task_struct` de cada tarea:

- `void *pmc`, es un puntero a datos específicos de los PMCs por hilo. Este puntero, `pmc`, aunque es declarado de tipo `void` en el parche de PMCTrack, se le asignarán datos mantenidos en la estructura `pmon_prof_t` incluida en fichero de cabecera `<pmc/mc_experiments.h>` de las fuente de PMCTrack.
- `unsigned char prof_enabled`, es un flag encargado de indicar si la monitorización esta activa sobre dicho proceso.

En primera instancia, se propuso como solución crear un parche del kernel con al menos el campo `void *pmc` incluido dentro del `task_struct`, para ser propuesto a la comunidad Linux y plantear su inclusión en el kernel Linux. Esto hubiera obligado a realizar un estudio para justificar la necesidad de tener un campo en el `task_struct` en el que se pueda guardar datos por proceso, ya no solo para PMCTrack sino también aplicable a otras necesidades. Esta idea fue descartada una vez se vio la posibilidad de mantener datos por hilo en el `task_struct` usando la abstracción del subsistema perf_events. Se ha de tener en cuenta que el uso de perf_events que se va a exponer en la siguientes subsecciones no es parte del uso para el que perf_events está diseñado. Para llegar a encontrar una forma de poder asignar un puntero arbitrario a un `task_struct` con perf_events, ha sido necesario un estudio en profundidad del subsistema de eventos perf_events (el cual dispone de una documentación escasa y en muchos casos inexistente).

Para entender cómo ha surgido la idea de la posibilidad de hacer uso de perf_events se procede a hacer una serie de subsecciones. Primero se explican las estructuras que componen un evento perf_events y cómo pueden ser aprovechadas para el uso que se pretende hacer en PMCTrack. A continuación se pondrá en conocimiento qué eventos de perf_events son los más adecuados para usar y la implementación que se ha llevado a cabo. Por último se muestra cómo se ha unificado el acceso a los datos de monitorización de PMCTrack, para adaptarse a esta nueva implementación.

4.3.1 Cómo asignar un puntero arbitrario a cada `task_struct` usando perf_events

En perf_events los contadores de rendimiento existentes se implementan por las denominadas PMUs del kernel, *Performance Monitoring Units*, representadas por la estructura `struct pmu`, definida por perf_events. Existen numerosas PMUs definidas pero las más relevantes, por su simplicidad y por el uso que se propone hacer, son aquellas que definen eventos *software*. Cada PMU además puede asociar a cada evento, `struct perf_event`, un puntero arbitrario en el campo `pmu_private` para que use como desee aquella PMU

que implementa el evento en cuestión:

```
struct perf_event {
    [...]
    struct pmu          *pmu;
    void                *pmu_private;
    [...]
}
```

Para poder aprovechar dicho campo de `perf_event` como portador de las datos de PMCTrack se ha de tener en cuenta que cada `task_struct` tiene asociada una lista de perf_events en el campo `perf_event_list`, que es una lista doblemente enlazada protegida por un mutex:

```
struct task_struct {
[...]
#ifdef CONFIG_PERF_EVENTS
    struct perf_event_context   *perf_event_ctxp[perf_nr_task_contexts];
    struct mutex                perf_event_mutex;
    struct list_head            perf_event_list;
#endif
[...]
}
```

De lo expuesto, surge la idea de que para toda tarea que se cree en el sistema se instale un evento ficticio por defecto, incorporado en la lista `perf_event_list` de su `task_struct`, que en su campo `pmu_private` lleve una estructura que contenga los datos de PMCTrack.

Esta lista se inicializa en `perf_event_init_task()`, definida en *kernel/events/core.c* del kernel, y que se invoca desde la función `copy_process()`, función que crea un nuevo proceso como copia del anterior pero no lo inicia aún. Se muestra a continuación el orden de invocación de las funciones dentro de `copy_process()` en el kernel:

```
static __latent_entropy struct task_struct *copy_process(
                    struct pid *pid,
                    int trace,
                    int node,
                    struct kernel_clone_args *args)
{
[...]
    /* Perform scheduler related setup. Assign this task to a CPU. */
    retval = sched_fork(clone_flags, p);
    if (retval)
        goto bad_fork_cleanup_policy;

    retval = perf_event_init_task(p);
    if (retval)
        goto bad_fork_cleanup_policy;
    retval = audit_alloc(p);
    if (retval)
```

```
        goto bad_fork_cleanup_perf;
    /* copy all the process information */
    shm_init_task(p);
    retval = security_task_alloc(p, clone_flags);
    if (retval)
        goto bad_fork_cleanup_audit;
    retval = copy_semundo(clone_flags, p);
    if (retval)
        goto bad_fork_cleanup_security;
[...]
}
```

La función `perf_event_init_task()` se encuentra justo después de la invocación a `sched_fork()`, función en la que se instala la callback **pmcs_alloc_per_thread_data** del parche de PMCTrack. La necesidad de que `perf_event_list` esté inicializada para poder insertar un evento hace que como función a trazar para la callback **pmcs_alloc_per_thread_data** con ftrace, se elija la función `copy_semundo()`, función que posee los argumentos necesarios para la callback.

La decisión de no usar la función `security_task_alloc()`, que se encuentra antes, se debe a su falta de disponibilidad para ser trazada con ftrace en arquitecturas ARM.

4.3.2 Incorporación de un evento *software* para los datos de PMCTrack

Conocida la posibilidad de usar el puntero arbitrario de una estructura `struct pmu` asociada a un evento perf_events, se procede a explicar que PMUs son apropiadas para usar como evento *software* y como crear dicho evento.

El primer paso es encontrar un evento *software* que se pueda asignar a cada `task_struct` por defecto. Mediante el comando `perf list`, es posible obtener el listado de eventos *software*:

```
$ perf list

List of pre-defined events (to be used in -e):

[...]
  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
```

```
  task-clock                                         [Software event]
[...]
```

Sus correspondientes IDs para poder configurar cualquiera de esos eventos desde dentro del kernel se encuentran en el fichero *include/uapi/linux/perf_event.h* de la fuentes de Linux[73]:

```
enum perf_sw_ids {
    PERF_COUNT_SW_CPU_CLOCK             = 0,
    PERF_COUNT_SW_TASK_CLOCK            = 1,
    PERF_COUNT_SW_PAGE_FAULTS           = 2,
    PERF_COUNT_SW_CONTEXT_SWITCHES      = 3,
    PERF_COUNT_SW_CPU_MIGRATIONS        = 4,
    PERF_COUNT_SW_PAGE_FAULTS_MIN       = 5,
    PERF_COUNT_SW_PAGE_FAULTS_MAJ       = 6,
    PERF_COUNT_SW_ALIGNMENT_FAULTS      = 7,
    PERF_COUNT_SW_EMULATION_FAULTS      = 8,
    PERF_COUNT_SW_DUMMY                 = 9,
    PERF_COUNT_SW_BPF_OUTPUT            = 10,

    PERF_COUNT_SW_MAX,                  /* non-ABI */
};
```

De todos estos eventos *software*, aquel que resulta más adecuado es el `SW_DUMMY`. Se trata de un evento de carácter provisional que no cuenta nada. Este evento ficticio permite recoger registros de muestras informativas sin necesidad de tener una PMU asociada [73]. Esto en particular es ideal para el tipo de `perf_event` ficticio que se pretende usar como portador de los datos de PMCTrack.

La creación de dicho evento se ha de realizar en la callback **pmcs_alloc_per_thread_data**, callback en la cual el módulo de kernel de PMCTrack crea, asigna memoria e inicializa la estructura `pmon_prof_t` con los datos de monitorización asignados a una tarea. Originalmente esta estructura se asignaba dentro de esta callback al campo `pmc` del `task_struct` añadido con el parche de PMCTrack. Esto implica que ahora la callback se tiene que encargar de crear el evento ficticio, asignarle los datos de la estructura `pmon_prof_t` ya creada y añadir dicho evento a la lista de perf_events del `task_struct` correspondiente.

Para crear un `perf_event` y asociarlo a una tarea desde el propio kernel se ha de usar la siguiente función de la API de perf_events, cuyos parámetros se describen a continuación:

```
struct perf_event *
perf_event_create_kernel_counter(struct perf_event_attr *attr, int cpu,
                 struct task_struct *task,
                 perf_overflow_handler_t overflow_handler,
                 void *context)
```

- **attr**, es la estructura `perf_event_attr` con los atributos del contador asocia-

do al evento a crear. En el caso del evento ficticio el tipo ha de ser *software*, `PERF_TYPE_SOFTWARE`, con información de configuración específica del tipo `PERF_COUNT_SW_DUMMY`. También es de interés que esté desactivado por defecto ya que será usado simplemente para el almacenamiento de datos.

```
struct perf_event_attr perf_sw_attr = {
        .type           = PERF_TYPE_SOFTWARE,
        .config         = PERF_COUNT_SW_DUMMY,
        .size           = sizeof(struct perf_event_attr),
        .pinned         = 1,
        .disabled       = 1,
};
```

- **cpu**, es la CPU a la que está vinculado el contador. Si se indica una CPU se le asocia al evento de rendimiento a su contexto. Esto no es el comportamiento deseado, el evento no tiene que estar fijado a una CPU sino que ha de ser accesible desde cualquier CPU por lo que se la ha de asignar el valor -1.
- **task**, es el `task_struct` al cual se quiere asignar el evento, en este caso con los datos de monitorización de PMCTrack. Este se corresponde con el `task_struct` que se obtiene en la callback **pmcs_alloc_per_thread_data** invocada cuando se hace un *fork* de un proceso.
- **overflow_handler** y **context**, no son necesarios para el uso que se va a dar al evento.

De tal forma, la creación del evento ficticio de perf_events dentro de PMCTrack resulta en la siguiente implementación:

```
struct perf_event *event = NULL;
[...]
event = perf_event_create_kernel_counter(&perf_sw_attr, -1, task, NULL, NULL);
```

Hay que tener en cuenta que `perf_event_create_kernel_counter()` se encarga de reservar la memoria para el `perf_event` creado, esto requiere que la callback **pmcs_alloc_per_thread_data** se tenga que llamar con expropiación habilitada.

Una vez creado el evento `SW_DUMMY` es momento de asignarle a su campo `pmu_private` los datos de PMCTrack. En un principio se barajaron distintas opciones en la elección de la estructura de datos encargada de mantener los datos de monitorización en el evento ficticio. Esto llevó a una implementación que permite unificar de la mejor forma posible las ramas de código de PMCTrack que usan el parche de PMCTrack y las que no. Esta implementación consiste en que el evento lleva en su campo `pmu_private`, como datos de PMCTrack, sólo la estructura `pmon_prof_t` definida por PMCTrack. Esto implica que ahora se ha de tener el flag `prof_enabled` dentro de esta misma estructura, junto con otros nuevos campos que se muestran a continuación:

```
#define SAFETY_CODE 0x7777
```

```
typedef struct {
#ifndef CONFIG_PMCTRACK
    int safety_control;
    unsigned char prof_enabled;
    struct perf_event *event;
    struct list_head links;         /* For the exited task list */
#endif
    [...]
    struct task_struct *this_tsk;   /* Backwards pointer to the task
                                       struct of this thread */
    [...]
} pmon_prof_t;
```

Cabe destacar que los campos `event` y `links` se emplean para la eliminación y acceso del evento ficticio, creado por PMCTrack dentro del `task_struct`, cuando dicho proceso alcanza la callback **pmcs_exit_thread**. Esto será explicado con detalle más adelante. El campo `safety_control` servirá como garantía de que el campo `pmu_private` al que se ha accedido es el correcto. Posteriormente, en la sección 4.3.3, se expondrá el conjunto de llamadas que se ha implementado en PMCTrack para la uniformización del acceso y recuperación de los datos de PMCTrack dentro del `task_struct`.

La inicialización de los nuevos campos de `pmon_prof_t` consiste simplemente en asignar los siguientes valores por defecto:

```
prof->safety_control = SAFETY_CODE;
prof->prof_enabled = 0;
prof->event=event;
event->pmu_private = prof;
```

Tras su inicialización, el puntero a la dirección de memoria de dicha estructura `pmon_prof_t` se asigna al campo `pmu_private` del evento ficticio recién creado en PMCTrack. Una vez se tiene el evento creado e inicializado por completo, se ha de insertar en la lista de perf_events del `task_struct` en cuestión como se muestra a continuación:

```
mutex_lock(&task->perf_event_mutex);
list_add_tail(&event->owner_entry, &task->perf_event_list);
mutex_unlock(&task->perf_event_mutex);
```

Se ha de tener en cuenta que esta lista doblemente enlazada de la tarea se encuentra protegida por un mutex, `perf_event_mutex`, incluido también como campo del propio `task_struct`.

Dado que se está añadiendo el evento justo después de que se llame a `perf_event_init_task()` –inicialización de la lista de perf_events– es seguro asumir que este evento se encontrará siempre en la primera posición de la lista. Aun así se toman medidas de prevención mediante la comprobación de la correcta correspondencia del campo `safety_control` cuando se realiza el acceso al evento en cuestión.

La creación del evento `SW_DUMMY`, la inicialización de los nuevos campos en `pmon_prof_t`, la asignación como datos privados del evento y la inserción del evento en el `task_struct` se han incluido en la nueva función de PMCTrack `create_pmctrack_task_event()`:

```
struct perf_event *create_pmctrack_task_event(struct task_struct* task,
                                              pmon_prof_t* prof)
```

Esta función será invocada por el módulo del kernel de PMCTrack cuando éste esté cargado en el sistema y cada vez que se realice el *fork* de un proceso.

En la versión sin parche de PMCTrack no es hasta la callback **pmcs_free_per_thread_data** que se elimina por completo los datos de monitorización de PMCTrack. Esto significa que en la callback **pmcs_exit_thread**, cuando un proceso pasa a estado `TASK_DEAD`, no se eliminan y se siguen manteniendo en el `task_struct`. Esto plantea un problema con la nueva instalación de la callback **pmcs_exit_thread** en el tracepoint de la función `do_exit()` del kernel, ya que en esta función se ejecuta `perf_event_exit_task()`.

```
void __noreturn do_exit(long code)
{
    [...]
    trace_sched_process_exit(tsk);
    [...]
    /*
     * Flush inherited counters to the parent - before the parent
     * gets woken up by child-exit notifications.
     */
    perf_event_exit_task(tsk);
    [...]
}
```

Se ha de tener en cuenta que dentro de `perf_event_exit_task()` la primera operación es un recorrido completo de la lista de perf_events del `task_struct` eliminando uno a uno cada evento que contiene. Esto implica que se pierde acceso al evento de datos de monitorización de PMCTrack antes de que se realice su `free_task()`, función del kernel en la que se encuentra instalada la callback **pmcs_free_per_thread_data**. Para solventar la pérdida temprana del evento, antes de terminar todos los posibles accesos, se ha creado dentro de PMCTrack una lista circular doblemente enlazada a la cual se ha llamado `gone_tasks` y cuyo acceso se protege con un *Reader–Writer spin lock*.

```
DEFINE_RWLOCK(gone_tasks_lock);
LIST_HEAD(gone_tasks);
```

Esta lista contendrá los datos de monitorización de PMCTrack de aquellos procesos que han realizado *exit*. Los elementos de esta lista de eventos se han hecho de tipo `pmon_prof_t`, de ahí la inclusión del nuevo campo `struct list_head links` dentro de la misma estructura `pmon_prof_t`, mencionado anteriormente.

Para la interacción con esta lista se han creado un conjunto de llamadas sencillas dentro

de PMCTrack. Éstas permiten el acceso, inserción y eliminación de elementos a la lista `gone_tasks` y son las siguientes: `get_prof_exited_task()`, `add_prof_exited_task()` y `del_prof_exited_task()`.

El nuevo comportamiento de PMCTrack, versión sin parche, en la callback **pmcs_exit_thread**, consiste en las siguientes operaciones:

```
pmon_prof_t *prof=get_prof(tsk);
[...]
/* Free up fake perf event */
perf_event_release_kernel(prof->event);
prof->event=NULL;
add_prof_exited_task(prof);
```

En el primer paso es necesario deshacerse por completo el evento `perf_event`, antes de que posteriormente sea eliminado de la lista de perf_events del `task_struct` en `perf_event_exit_task()`. Esto se consigue mediante la llamada a la función `perf_event_release_kernel()` de la API de perf_events, para ello se necesita un puntero referencia a dicho evento, es por esto que se ha añadido el campo nuevo al `pmon_prof_t` con dicha referencia, `struct perf_event *event`. Tras la liberación del evento se asigna a `NULL` su referencia en los datos de PMCTrack para evitar accesos a memoria indeseados.

Una vez liberado el evento ya se está en condición de guardar los datos, `pmon_prof_t`, en `gone_tasks` mediante `add_prof_exited_task()`. El hecho de que a partir de esta callback los datos se encuentren en `gone_tasks` implica que si se pretende hacer acceso a los datos de monitorización de una tarea que ha sido ya marcada como `TASK_DEAD`, este acceso se debe realizar dentro de la nueva lista. Esto será explicado en detalle en la siguiente sección 4.3.3.

La eliminación del evento usado por PMCTrack se realiza al efectuar la callback **pmcs_free_per_thread_data**. Esto implica que los datos de monitorización asociados al evento ya se encuentran en disposición de la lista `gone_tasks` y dicho evento ya ha sido eliminado anteriormente. Para ello se hace uso de la una nueva función `del_prof_exited_task()`, encargada de eliminar el `pmon_prof_t` dado de la lista `gone_tasks`.

4.3.3 Interacción con un evento *software* en el `task_struct` con los datos de PMCTrack

Con la incorporación de los nuevos campos a `pmon_prof_t` se ha establecido que PMCTrack pasa a acceder a todos los datos de monitorización por proceso a través de `pmon_prof_t`. De este modo se unifica la forma en que se accede a los datos entre la versión de PMCTrack con parche del kernel y sin él. El hecho de haber realizado esta unificación permite evitar el uso intensivo de sentencias de compilación condicional en las fuentes de PMCTrack. Si por el contrario se hubiera optado por el uso de sentencias

de compilación condicional esto habría dificultado la legibilidad del código fuente de la herramienta y supuesto la necesidad por parte de los desarrolladores de conocer las diferentes maneras de acceso a los datos.

Para cumplir esta unificación se ha creado un conjunto reducido de nuevas llamadas para garantizar que al usar la llamada a funciones de acceso, estas accederán de la manera adecuada según la versión de PMCTrack sobre la que se esté ejecutando. Este conjunto de llamadas de acceso consta de las siguientes tres funciones:

```
static inline pmon_prof_t* get_prof(struct task_struct* p) { ... }
static inline unsigned char get_prof_enabled(pmon_prof_t *prof) { ... }
static inline void set_prof_enabled(pmon_prof_t *prof, unsigned char enable) { ... }
```

- La función `get_prof()` obtiene los datos de monitorización, `pmon_prof_t`, del `task_struct` de la tarea dada.
- La función `get_prof_enabled()` indica el valor del flag `prof_enabled` –a partir de los datos de monitorización de una tarea– para conocer si se encuentra la monitorización activa.
- La función `set_prof_enabled()` establece un nuevo valor al flag `prof_enabled` de una tarea a través de los datos de monitorización de dicha tarea.

Estas tres funciones son incluidas y declaradas en el fichero de cabecera `mc_experiments.h` del código fuente de PMCTrack, fichero en el que también se encuentra definida la estructura `pmon_prof_t`. Esto significa que cualquier fichero de PMCTrack que accediese antes del cambio a `pmon_prof_t` ya se encuentra en disposición de uso de este nuevo conjunto de llamadas.

La más compleja de ellas se puede considerar `get_prof`, debido al nuevo planteamiento de recuperación de los datos de monitorización de un `task_struct` en la versión de PMCTrack sin parche. Como se ha expuesto anteriormente, los datos de PMCTrack por proceso se pueden encontrar almacenados en dos lugares distintos según si el proceso al que estén asociados haya realizado *exit* –lo que implica que sus datos se han pasado a almacenar dentro de `gone_tasks`– o por el contrario el proceso aún no haya efectuado *exit* –entonces sus datos aún se mantienen en el campo `pmu_private` de su `perf_event` dentro de la lista de perf_events del `task_struct` de dicho proceso–.

En el caso de que aún no se haya ejecutado la callback **pmcs_exit_thread** sobre el proceso, no ha realizado *exit*, el acceso es directo a través de su `task_struct`. Debido a que es seguro asumir que el `perf_event` creado para PMCTrack se encuentra en la primera posición, se intenta obtener consultando siempre la primera posición de la lista del `task_struct` dado de la siguiente forma:

```
event = list_first_entry(&(p->perf_event_list), typeof(*event), owner_entry);
if (event != NULL)
    prof = (pmon_prof_t*)(event->pmu_private);
```

Aun así por seguridad se comprueba que dicho evento exista y se recupera su campo

pmu_private como estructura pmon_prof_t, en la cual también se ha de garantizar que es el evento correcto comprobando su SAFETY_CODE como se muestra a continuación:

```
if (!prof || prof->safety_control != SAFETY_CODE) {
    printk(KERN_INFO "Failed perf_event safety check\n");
    return NULL;
}
```

En el acceso dentro de la lista al primer elemento, a través de list_first_entry(), se obvia la protección con mutex de ésta. La decisión de este acceso no seguro se hace como salvaguarda de un posible bloqueo completo del sistema ya que, aunque se exige al usuario que se abstenga del uso de perf durante el uso de PMCTrack, en caso contrario de que se accediera de manera bloqueante a la lista de perf_events del task_struct desde el espacio de usuario, provocaría un congelamiento infinito de la máquina debido a la gravedad del bloqueo que puede causar en el planificador del kernel.

Para el caso de que ya se haya ejecutado la callback **pmcs_exit_thread** sobre el proceso, ha realizado *exit*, el acceso, en la función get_prof(), se debe hacer a través de get_prof_exited_task() de la siguiente manera:

```
if ((p->state & TASK_DEAD) != 0)
    return get_prof_exited_task(p);
```

Esta función buscará el pmon_prof_t, correspondiente al task_struct indicado, dentro de la lista gone_tasks.

Por el contrario para la versión de PMCTrack con parche del kernel, get_prof() mantiene el acceso a los datos de monitorización –mantenidos en la estructura pmon_prof_t– mediante el campo pmc del task_struct incorporado en el parche de PMCTrack. Esto se logra con el siguiente sencillo acceso:

```
return (p == NULL ? NULL : (pmon_prof_t*) p->pmc);
```

Se ha de tener en cuenta que el acceso al flag prof_enabled deja de ser a través del task_struct y se unifica a un acceso siempre mediante una estructura pmon_prof_t dada, utilizando la nueva llamada definida get_prof_enabled(). Por otro lado, se ha creado una nueva llamada para la modificación del prof_enabled, a la cual se ha llamado set_prof_enabled(). Dicha función pasa a hacer también la modificación a través de una estructura pmon_prof_t dada.

Para PMCTrack sin parche del kernel ambas llamadas hacen el acceso al flag directamente por el nuevo campo prof_enabled de pmon_prof_t. De esta forma, el acceso y modificación de prof_enabled en la función set_prof_enabled(), se realiza como se muestra a continuación:

```
if (prof) prof->prof_enabled = enable;
```

En PMCTrack con parche del kernel el acceso y modificación de prof_enabled pasa a ser a través del puntero referencia al propio task_struct del correspondiente pmon_prof_t.

Este puntero se mantiene en el campo `this_tsk` de la estructura `pmon_prof_t`. Desde dicho puntero se puede acceder al campo `prof_enabled` del `task_struct` instalado en el parche de PMCTrack. Este acceso tiene la siguiente codificación para `set_prof_enabled()`:

```
if (prof) prof->this_tsk->prof_enabled = enable;
```

4.4 Alternativa para encontrar un proceso por PID en PMCTrack

La última parte del parche de PMCTrack que queda por explicar es la exportación de la función `find_process_by_pid()` dentro del kernel Linux, función encargada de encontrar un proceso con un valor PID equivalente al indicado.

La necesidad de uso de `find_process_by_pid()` viene, entre otros usos, principalmente del modo *attach* de PMCTrack (opción `-p` de la línea de comandos). Este modo permite a un proceso que se encuentre ya en ejecución abrir una sesión de monitorización con PMCTrack estableciendo la configuración de eventos indicada en la línea de comandos. Al igual que ya se ha expuesto para otras partes del parche de PMCTrack, también se barajó la opción de crear un parche para el kernel Linux con la exportación de dicha función y solicitar a la comunidad Linux su inclusión en la distribución oficial del kernel Linux. Esta petición parece razonable ya que encontrar un proceso por su `pid` es una funcionalidad de gran utilidad para cualquier tipo de aplicación relacionada con el kernel Linux. Aun así, por la naturaleza de las llamadas que se hacen dentro de `find_process_by_pid()`, se determinó una solución alternativa que no implica la aceptación y mantenimiento de un parche en la distribución oficial del kernel Linux.

Esta solución consiste en crear una función *wrapper* propia de PMCTrack a la que se ha llamado `pmctrack_find_process_by_pid()` que replica el comportamiento de `find_process_by_pid()` dentro del kernel Linux. La reproducción de su funcionalidad es posible ya que en el árbol de llamadas que se hace dentro de `find_process_by_pid()` se llega en cualquier caso a una función que es exportada dentro del kernel. La definición de la nueva función `pmctrack_find_process_by_pid()` se realiza, dentro de PMCTrack, en el fichero cabecera `mc_experiments.h` y su implementación, que se muestra a continuación, en el correspondiente fichero de código `mc_experiments.c`:

```
extern struct pid *find_pid_ns(int nr, struct pid_namespace *ns);
extern struct task_struct *pid_task(struct pid *pid, enum pid_type type);
extern struct pid_namespace *task_active_pid_ns(struct task_struct *tsk);

struct task_struct *pmctrack_find_process_by_pid(pid_t pid)
{
    if (pid) {
        struct pid_namespace *ns = task_active_pid_ns(current);
```

```
        RCU_LOCKDEP_WARN(!rcu_read_lock_held(),
             "find_task_by_pid_ns() needs rcu_read_lock() protection");
        return pid_task(find_pid_ns(pid, ns), PIDTYPE_PID);
    } else {
        return current;
    }
}
```

Aunque esta solución implica un mantenimiento por los desarrolladores de PMCTrack por posibles modificaciones en futuras versiones del kernel Linux, puede asumirse que en funciones de esta naturaleza dentro del kernel Linux su implementación se mantenga sin cambios profundos.

4.5 Integración dentro del código fuente de PMCTrack

Como se ha mencionado al principio este capítulo (4), la nueva implementación de la API de PMCTrack sin parche del kernel Linux se ha realizado de forma que se encuentre incluida dentro de cada propio módulo del kernel de PMCTrack compilado. La implementación de las callbacks de la API del kernel de PMCTrack con ftrace y tracepoints se ha realizado en un nuevo fichero `pmctrack_stub.c`, con su correspondiente fichero de cabecera `pmctrack_stub.h`. En este último fichero se encuentra la definición de la misma interfaz de callbacks que para la API del kernel de PMCTrack con parche del kernel, `pmc_ops_t`:

```
typedef struct __pmc_ops {
    int     (*pmcs_alloc_per_thread_data)(unsigned long,struct task_struct*);
    void    (*pmcs_save_callback)(void* prof, int);
    void    (*pmcs_restore_callback)(void* prof, int);
    void    (*pmcs_tbs_tick)(void* p, int);
    void    (*pmcs_exec_thread)(struct task_struct*);
    void    (*pmcs_free_per_thread_data)(struct task_struct*);
    void    (*pmcs_exit_thread)(struct task_struct*);
    int     (*pmcs_get_current_metric_value)(struct task_struct* task, int key,
                                             uint64_t* value);
} pmc_ops_t;
```

La creación del evento perf_events, como portador de los datos de monitorización por proceso de PMCTrack, se ha realizado sobre el fichero ya existente `mchw_core.c`, parte central del código independiente de la plataforma de PMCTrack. El conjunto de llamadas para el acceso y modificación de dicho evento, se han incluido convenientemente en el fichero de cabecera `mc_experiments.h` de PMCTrack, ocultando así los detalles de implementación. De esta manera todos los ficheros que ya hacen uso de `pmon_prof_t` tiene acceso al nuevo conjunto de llamadas sobre este.

En esta nueva versión de PMCTrack, la API del kernel de PMCTrack pasa a ser incluida en `mchw_core.c`, núcleo de PMCTrack, a través de `<pmc/pmctrack_stub.h>`. Es esta

nueva inclusión la que se encarga de determinar que versión de la API del kernel de PMCTrack se ha de utilizar en función de si existe o no el parche de PMCTrack en el kernel sobre el que se encuentra funcionando el sistema.

Por otro lado se han hecho modificaciones en la compilación y la carga del módulo del kernel de PMCTrack para ofrecer compatibilidad hacia atrás con módulos de PMCTrack con backends distintos al de perf_events. Estas modificaciones son realizadas en el script de espacio de usuario `pmctrack-manager`, este script permite realizar la compilación, limpieza y carga de los distintos módulos del kernel de PMCTrack con backends compatibles con el sistema.

El script `pmctrack-manager` ahora ha pasado a detectar si en el kernel Linux en el que está siendo ejecutado no se encuentra presente el parche de PMCTrack. En este caso la opción `build` realiza automáticamente la compilación del módulo de PMCTrack con el backend de perf_events (mchw_perf), y la opción `load-module` instala directamente ese módulo del kernel de PMCTrack.

Para ofrecer la opción de usar un módulo de PMCTrack con otro backend distinto al de perf_events con un kernel sin el parche de PMCTrack, se ha creado la variable de entorno `FORCE_LEGACY`. Esta variable de entorno fuerza a la opción ejecutada con `pmctrack-manager` a considerar otros módulos compatibles. En el caso de la opción `build` se compilarán módulos del kernel de PMCTrack con el resto de backends que detecte compatibles para el sistema. Para la opción `load-module` con `FORCE_LEGACY` establecida, se le permitirá elegir al usuario entre los módulos compatibles que han sido compilados, tal como se muestra a continuación:

```
$ FORCE_LEGACY=yes pmctrack-manager load-module
Several compatible modules found.
1) intel-core
2) core2
3) perf
Please pick one [1-3]
```

Capítulo 5

Extensión del backend de perf_events

Con la publicación de la versión v1.5 de PMCTrack se introdujo un nuevo backend experimental implementado haciendo uso del subsistema perf_events de Linux [7]. Como ya se ha comentado, la implementación de este backend demostró que es posible hacer una integración entre perf_events y PMCTrack. La idea de este backend genérico es hacer uso del subsistema de eventos perf_events desde el kernel para el acceso a los contadores de monitorización del rendimiento en aquellas arquitecturas y modelos de procesadores que tengan soporte en perf_events.

Este nuevo backend experimental no se implementó de forma completa y tiene limitaciones importantes. La limitación más grave es que sólo soporta procesadores Intel x86. Esto implica que, aunque sirve como prueba de concepto, no ofrece un backend completamente genérico. El backend experimental también presenta una limitación importante en la cantidad de eventos de perf_events soportados, solamente permite el uso de eventos que corresponden a contadores *hardware*. Como se ha expuesto en la sección 3.5, perf_events ofrece una gran variedad de eventos para monitorizar rendimiento que no provienen de los contadores *hardware*, a los que no se tiene acceso dentro del backend experimental de perf_events (p.ej. consumo de energía, ocupación del último nivel de caché, etc).

En este capítulo se procede a explicar los cambios que se han realizado en el backend de perf_events de PMCTrack. No obstante antes de poder introducir estas modificaciones se presenta a bajo nivel la estructura de los backends de PMCTrack. El capítulo se estructura en las siguientes secciones. En la **sección 5.1** se muestra cómo el subsistema perf_events del kernel expone la información de eventos y PMUs mediante el sysfs. En la **sección 5.2** se describe a bajo nivel la estructura de un backend en PMCTrack. En la **sección 5.3** se explica cómo se ha llevado a cabo la generalización del backend de perf_events para que funcione en distintas arquitecturas, así como la ampliación del soporte de eventos de perf_events disponibles a monitorizar con PMCTrack a través de este backend. En la **sección 5.4** se expone los cambios realizados en la nueva implementación de la API del kernel de PMCTrack para hacerla compatible con arquitecturas ARM. Finalmente, en la **sección 5.5** se explica la implementación necesaria que se ha realizado para que

la nueva API de PMCTrack también tenga soporte de uso con los backends *legacy* de PMCTrack.

5.1 Eventos que perf_events expone en espacio de usuario mediante sysfs

El subsistema de eventos perf_events cuenta con una extensa colección de eventos de monitorización que varía según el modelo de procesador. Para tener acceso a la información del subsistema de eventos perf_events, éste usa sysfs[1] y la herramienta de línea de comandos `perf` para exponer la información de configuración de los eventos al espacio de usuario.

Desde Linux 2.6.34, el kernel soporta tener múltiples PMUs disponibles para su monitorización. El subsistema de eventos perf_events expone a través de sysfs la información sobre cómo programar las PMUs que existen en el sistema. Esta información se puede encontrar en */sys/bus/event_source/devices/*, donde cada subdirectorio corresponde a una PMU diferente [73], [74]:

```
$ ls  /sys/bus/event_source/devices/
breakpoint   kprobe      uncore_cbox_0  uncore_cbox_5  uncore_r2pcie
cpu          msr         uncore_cbox_1  uncore_cbox_6  uncore_r3qpi_0
cstate_core  power       uncore_cbox_2  uncore_cbox_7  uncore_r3qpi_1
cstate_pkg   software    uncore_cbox_3  uncore_ha_0    uncore_ubox
intel_pt     tracepoint  uncore_cbox_4  uncore_pcu     uprobe
```

Cada subdirectorio de las PMUs contiene (desde Linux 2.6.38) un fichero */sys/bus/event_source/devices/<pmu_name>/type*. Este fichero contiene un número entero que puede utilizarse en el momento de crear la configuración de un evento para indicar en el campo `type`, de la estructura `perf_event_attr` del kernel, que PMU se desea utilizar.

```
$ cat /sys/bus/event_source/devices/power/type
10
$ cat /sys/bus/event_source/devices/cpu/type
4
$ cat /sys/bus/event_source/devices/software/type
1
```

Algunos de los subdirectorios de las PMUs además contienen (desde Linux 3.4) otro subdirectorio */sys/bus/event_source/devices/<pmu_name>/format/*. Este subdirectorio muestra información sobre los subcampos específicos de cada arquitectura disponibles para programar los diversos campos de configuración que contiene la estructura `perf_event_attr`. El contenido de cada fichero se compone del nombre del campo de

[1] sysfs es un pseudo-sistema de archivos proporcionado por el kernel Linux que exporta al espacio de usuario información sobre varios subsistemas del kernel, dispositivos *hardware*, módulos del kernel, sistemas de archivos y otros componentes del kernel.

`perf_event_attr` (`config`, `config1` o `config2`) seguido de dos puntos y de una serie de rangos de bits enteros separados por comas.

Por ejemplo el fichero *ldlat* puede contener el valor `config1:0-15` que indica que `ldlat` es un atributo que ocupa los bits del 0 al 15 del campo `config1` de la estructura `perf_event_attr`:

```
$ cat /sys/bus/event_source/devices/power/format/event
config:0-7
$ cat /sys/bus/event_source/devices/cpu/format/ldlat
config1:0-15
$ cat /sys/bus/event_source/devices/cpu/format/umask
config:8-15
```

Para eventos con PMU de tipo tracepoint la configuración que se le ha de indicar al campo `config` de `perf_event_attr` del evento se encuentra en el fichero */sys/kernel/debug/tracing/events/<tracepoint_source>/<tracepoint_name>/id*. La configuración de estos eventos solo estará disponible si ftrace está presente en el kernel.

```
$ cat /sys/kernel/debug/tracing/events/power/cpu_frequency/id
459
```

Dentro de cada subdirectorio de las PMUs también se encuentran el subdirectorio *events*, */sys/bus/event_source/devices/<pmu_name>/events/*. Este subdirectorio contiene ficheros con eventos predefinidos.

```
$ ls /sys/bus/event_source/devices/power/events/
energy-pkg        energy-pkg.unit  energy-ram.scale
energy-pkg.scale  energy-ram       energy-ram.unit
```

No se trata necesariamente de listas completas de todos los eventos soportados por una PMU, sino normalmente de un subconjunto de eventos considerados básicos o útiles. La herramienta `perf` se refiere a estos como eventos tipo `[Kernel PMU event]`. Si alguno de estos eventos se muestran en una unidad determinada entonces también existen ficheros para mostrar la unidad y la escala.

El contenido de estos ficheros son cadenas que describen la configuración de los eventos. Estas cadenas se expresan en términos de los campos que se encuentran en el directorio *../format/* mencionado anteriormente. En concreto cada archivo contiene una lista de nombres de atributos separados por comas. Cada entrada tiene un valor opcional (hexadecimal o decimal). Si no se especifica ningún valor, se supone que es un campo de un solo bit con un valor de 1. Como ejemplo, se muestra a continuación las cadenas que describen la configuración de los eventos de consumo energético global y de cargas en la memoria.

```
$ cat /sys/bus/event_source/devices/power/events/energy-pkg
event=0x02
$ cat /sys/bus/event_source/devices/cpu/events/mem-loads
event=0xcd,umask=0x1,ldlat=3
```

También es posible acceder a la información de estos ficheros de eventos a través de la ruta */sys/devices/<event_source>/events* del sistema.

```
$ ls /sys/devices/*/events
/sys/devices/cpu/events:
branch-instructions  el-abort      ref-cycles                      tx-abort
branch-misses        el-capacity   topdown-fetch-bubbles           tx-capacity
bus-cycles           el-commit     topdown-recovery-bubbles        tx-commit
cache-misses         el-conflict   topdown-recovery-bubbles.scale  tx-conflict
cache-references     el-start      topdown-slots-issued            tx-start
cpu-cycles           instructions  topdown-slots-retired
cycles-ct            mem-loads     topdown-total-slots
cycles-t             mem-stores    topdown-total-slots.scale

/sys/devices/cstate_core/events:
c3-residency  c6-residency  c7-residency

/sys/devices/cstate_pkg/events:
c2-residency  c3-residency  c6-residency  c7-residency

/sys/devices/msr/events:
aperf               cpu_thermal_margin.snapshot  mperf  tsc
cpu_thermal_margin  cpu_thermal_margin.unit      smi

/sys/devices/power/events:
energy-pkg        energy-pkg.unit  energy-ram.scale
energy-pkg.scale  energy-ram       energy-ram.unit
```

El conocimiento de la existencia de esta funcionalidad de perf_events de exponer los eventos en el sysfs y la posibilidad de usar los valores de configuración expuestos para programar los campos de la estructura `perf_event_attr` de un evento desde el kernel, es resultado del estudio exhaustivo del código fuente del subsistema perf_events. Ha sido necesario un estudio a bajo nivel de perf_events debido la escasa documentación disponible sobre el subsistema perf_events.

Por otro lado la herramienta de línea de comandos `perf`, también ofrece información detallada sobre la configuración de los eventos no genéricos que componen el subsistema de eventos perf_events a través de la opción `list --details` [62]:

```
$ perf list --details

List of pre-defined events (to be used in -e):

  branch-instructions OR branches                     [Hardware event]
[...]
  alignment-faults                                    [Software event]
[...]
  L1-dcache-load-misses                               [Hardware cache event]
[...]
  branch-instructions OR cpu/branch-instructions/     [Kernel PMU event]
  branch-misses OR cpu/branch-misses/                 [Kernel PMU event]
```

```
  bus-cycles OR cpu/bus-cycles/                        [Kernel PMU event]
[...]
cache:
  l1d.replacement
       [L1D data line replacements]
        cpu/umask=0x1,(null)=0x1e8483,event=0x51/
[...]
memory:
  hle_retired.aborted
       [Number of times HLE abort was triggered (PEBS) (Precise event)]
        cpu/umask=0x4,(null)=0x1e8483,event=0xc8/
[...]
other:
  cpl_cycles.ring0
       [Unhalted core cycles when the thread is in ring 0]
        cpu/umask=0x1,(null)=0x1e8483,event=0x5c/
[...]
pipeline:
  arith.fpu_div_active
       [Cycles when divider is busy executing divide operations]
        cpu/umask=0x1,(null)=0x1e8483,event=0x14/
[...]
uncore cache:
  llc_misses.code_llc_prefetch
       [LLC prefetch misses for code reads. Derived from
        unc_c_tor_inserts.miss_opcode. Unit: uncore_cbox]
        uncore_cbox_6/umask=0x3,event=0x35,filter_opc=0x191/
[...]
uncore power:
  unc_p_clockticks
       [PCU clock ticks. Use to get percentages of PCU cycles events. Unit:
        uncore_pcu]
        uncore_pcu/event=0/
[...]
virtual memory:
  dtlb_load_misses.miss_causes_a_walk
       [Load misses in all DTLB levels that cause page walks Spec update:
        BDM69]
        cpu/umask=0x1,(null)=0x186a3,event=0x8/
[...]
```

El comando `list --details` muestra primero los eventos generalizados que ofrece el subsistema de eventos perf_events que pueden no tener soporte en todos los sistemas. Entre ellos están los `[Kernel PMU event]` cuya configuración se encuentra en los directorios de sysfs que se han mostrado anteriormente. A continuación de los eventos generalizados se muestran eventos específicos del sistema junto con una breve descripción y los valores de su configuración para perf_events.

El backend de perf_events de PMCTrack se creó para usar solo los eventos asociados a contadores *hardware*; contadores de los que ya se tenía conocimiento de la configuración

necesaria gracias al estudio realizado para el resto de backends de PMCTrack. Esto hace que el backend este limitado en la cantidad de eventos de perf_events a monitorizar. Tras el meticuloso estudio del subsistema perf_events se ha logrado obtener conocimiento de que perf_events expone la información de sus eventos al espacio de usuario a través de sysfs. Este nuevo conocimiento planteó la posibilidad de usar esta información desde el espacio del kernel por PMCTrack para ampliar los eventos disponibles a monitorizar.

5.2 Diseño de backends para PMCTrack

PMCTrack ha sido diseñado desde sus inicios siguiendo una estrategia modular, que se refleja en el diseño de los backends de PMCTrack. Los backends de PMCTrack han de implementar una serie de funciones definidas que permiten al núcleo de PMCTrack independiente de la plataforma (fichero *mchw_core.c* de las fuentes de PMCTrack) interactuar con el código específico de la plataforma. Es por esto que la implementación de la mayoría de estas funciones es específica de la plataforma, sus declaraciones se encuentran en el fichero *mc_experiments.h* de las fuentes de PMCTrack.

Las funciones que ha de implementar un backend de PMCTrack son las siguientes:

- `init_pmu()`

  Inicializa las propiedades de las PMUs de las distintas CPUs del sistema. Emplea mecanismos de descubrimiento de características del *hardware* dependientes de la arquitectura, para detectar las PMUs disponibles en el sistema. Las propiedades de cada PMU se almacenan en la estructura `pmu_props_t` de PMCTrack.

- `pmu_shutdown()`

  Detiene las PMUs y libera los recursos asignados por `init_pmu()`.

- `parse_pmcs_strconfig()`

  Se encarga de analizar una cadena de configuración de PMCs especificada en formato *raw*. Este formato *raw* de PMCTrack es una representación de bajo nivel orientada a escribir en los registros *hardware* (p.ej. `pmc0=0xc0,pmc1=0x76,...`). Durante el análisis de dicha cadena de configuración se rellenan los distintos campos de una estructura de tipo `pmc_usrcfg_t` en una lista cuya posición corresponde al número de contador a configurar. El tipo de datos `pmc_usrcfg_t` se ha definido dentro de PMCTrack para contener los distintos valores de los flags de configuración que se encuentran en cada contador de monitorización del rendimiento. La definición de esta estructura es específica de la plataforma; esto hace que el procesamiento asociado a los campos de `pmc_usrcfg_t` también deba ser específico de la plataforma.

  Esta función también proporciona el número de PMCs utilizados, una máscara de bits con el conjunto de PMCs utilizados por la cadena de configuración y otra

información relevante para el núcleo independiente de la plataforma de PMCTrack (*mchw_core.c*).

- `do_setup_pmcs()`

  Transforma la lista de configuraciones de los contadores dependiente de la plataforma creada por `parse_pmcs_strconfig()` en una estructura de bajo nivel. Esta estructura definida por PMCTrack es `core_experiment_t`. Contiene los datos necesarios para configurar los contadores *hardware*, o los eventos en el caso del backend de perf_events.

  En el caso concreto del backend de perf_events, `do_setup_pmcs()` se encarga también de la creación de los eventos de perf_events. Estos se crean deshabilitados y se mantienen en la estructura `core_experiment_t` para su posterior habilitación.

- `do_count_on_overflow()`

  A diferencia del resto, esta función no es implementada por el backend. La función `do_count_on_overflow()` ha de ser invocada desde el backend por el código de las PMUs cuando se gestiona una interrupción por desbordamiento de un PMC. Cuando se usa el modo de monitorización EBS (*Event Based Sampling*), esta función se encargará de leer los contadores de rendimiento e insertar una muestra del contador de rendimiento en el buffer de muestras de PMCTrack.

- `reset_overflow_status()`

  Si la plataforma dispone de un registro de desbordamiento del PMC éste se restablece en la CPU donde se invoca la función. Esta función no se implementa en el backend de perf_events.

- `mc_clear_all_platform_counters()`

  Realiza una inicialización por defecto de todos los PMCs en la CPU actual. Esta función tampoco se implementa en el backend de perf_events.

Aparte de estas funciones generales para todos los backends, para el backend de perf_events además se tuvieron que crear algunas funciones adicionales. Estas funciones no han sufrido ningún cambio y son las siguientes:

- `perf_enable_counters()`: habilita el contador de perf_events asociado a un `perf_event` dado.
- `perf_disable_counters()`: desactiva el contador de perf_events asociado a un `perf_event` dado.

El flujo de ejecución de las funciones que componen un backend comienza desde la carga del modulo de PMCTrack dentro del kernel. Cuando se produce la carga del módulo de PMCTrack se ejecuta la función `init_pmu()`. Ésta lleva a cabo un recorrido de todas las CPUs y sobre cada una de ellas ejecuta las operaciones de descubrimiento para conocer

qué tipo de *core*, el modelo, cuántos contadores tiene de cada tipo, la anchura en bits de los contadores, etc. Con la información de todas las CPUs y sus PMUs recabada se construye un array con estructuras `pmu_props_t` para ser usado por PMCTrack. La información recogida puede ser visualizada desde la entrada */proc/pmc/info* del sistema creada por PMCTrack. Ésta muestra la arquitectura y modelo de procesador junto con información sobre las PMUs detectadas.

Tras ejecutar la herramienta de línea de comandos `pmctrack` con una configuración de los contadores específica a través de la opción `-c`, dicha configuración se convierte a un formato *raw*. Para ilustrar esta conversión a formato *raw* de PMCTrack se considera el siguiente comando de ejemplo `pmctrack -c instr,llc_misses ./foo`. Este comando proporciona al usuario el número de instrucciones retiradas y fallos de la caché de último nivel (LLC) cada segundo. La conversión a formato *raw* de PMCTrack, para conseguir la representación de bajo nivel orientada a escribir en los registros *hardware*, se correspondería a `pmc0,pmc3=0x2e,umask3=0x41`. También es posible indicarle directamente a la herramienta de línea de comandos `pmctrack` una configuración en formato *raw*.

La función `parse_pmcs_strconfig()` se encarga de realizar el análisis de dicha cadena de configuración en formato *raw* para rellenar una estructura `pmc_usrcfg_t` por cada contador a usar. Tras un análisis correcto, por parte de `parse_pmcs_strconfig()`, de las opciones de los contadores que se le indican a PMCTrack, se configuran e inicializan las PMUs a usar con `do_setup_pmcs()`.

Si la monitorización se indica como modo EBS entonces `do_count_on_overflow()` será ejecutado cada vez que un evento alcance un recuento determinado.

Al descargar el módulo del kernel de PMCTrack del sistema, `pmu_shutdown()` libera toda la memoria reservada para el *parsing* de la configuración de las PMUs.

La implementación de estas funciones para el backend de perf_events se encuentra en el fichero *pmu_config_perf.c* de las fuentes de PMCTrack.

5.3 Modificaciones del backend de perf_events de PMCTrack

Como ya se ha mencionado, el backend de perf_events se creó para usar solo los eventos asociados a contadores *hardware* de rendimiento en arquitecturas Intel x86. Esto deja fuera el soporte a una gran cantidad de eventos que ofrece perf_events que provienen de otras PMUs (p.ej. cambios de contexto, número de fallos de página, etc).

Siguiendo el orden de ejecución de las funciones que componen el backend de perf_events (explicado en la sección anterior), se procede a hacer una descripción general de los cambios necesarios realizados en el backend de perf_events. Estos cambios se han realizado en el proyecto para dar soporte a procesadores ARM y AMD y expandir

la colección de eventos de perf_events a la que se tiene acceso con el backend de perf_events.

5.3.1 Descubrimiento de CPUs y PMUs asociadas

En el código del nuevo backend, se han añadido tablas con los nuevos modelos de procesador soportados. Estas tablas incluyen para cada modelo un identificador y una cadena de caracteres con el nombre correspondiente de la familia del procesador. Esta información es necesaria para poder hacer la correspondencia con el identificador de la CPU adquirido del *hardware* y detectar así el modelo dentro del backend permitiendo conocer sus PMUs.

Para ARM64 se ha añadido soporte para los modelos de *core* Cortex-A53, Cortex-A73, Cortex-A57. Para ARM32 el soporte se ha añadido para los modelos Cortex-A15, Cortex-A7. Esto también implica soporte para sistemas multicore asimétricos, aquellos con procesadores ARM *big.LITTLE* [75].

En cuanto a la compatibilidad con procesadores AMD dentro del backend de perf_events, se ha incorporado soporte para las familias de procesadores EPYC™ 7002 *Series Processors* [76] y Opteron [77] (ambas CPUs orientadas al uso en servidores).

Cuando se carga el módulo de PMCTrack en el sistema, se procede al descubrimiento de las CPUs y PMUs asociadas que dispone el sistema. Una gran limitación con la que cuenta perf_events es que su API del kernel no dispone de mecanismos de consulta de las características de las CPUs. La API de perf_events está orientada a eventos, no a contadores de monitorización del rendimiento. Estos contadores son abstraídos en eventos junto con otros eventos que no provienen de contadores *hardware* (véase sección 3.5), de esta forma se ocultan como tal en la API de perf_events. Dado que perf_events no permite en espacio del kernel conocer cuántos contadores *hardware* existen en la plataforma o sus propiedades, es necesario hacer uso de llamadas del kernel que permiten leer registros específicos del *hardware* o bien directamente con instrucciones ensamblador (p.ej. `CPUID` en x86).

Los mecanismos de descubrimiento usados por el backend se encapsulan en la función `init_pmu()`. Dentro de esta función se invoca a las funciones `init_pmu_props()` y `init_pmu_props_cpu()` las cuales son dependientes de la arquitectura. Es por esta dependencia que se han creado versiones nuevas de ambas funciones para ARM dentro del backend y se han modificado las ya presentes para x86 para añadir soporte con AMD.

En el caso de arquitecturas **ARM** se ha integrado el código de los backends originales de PMCTrack para ARM, con los mecanismos de consulta ya implementados –con el acceso a los registros de lectura específicos del *hardware*–. Estos mecanismos permiten averiguar el modelo del procesador y hacer la correspondencia de PMUs y sus propiedades en procesadores ARM. Aun así se han realizado pequeñas modificaciones sobre este código

heredado para incluir nuevos flags de configuración correspondientes a los eventos de perf_events. Estos flags permiten indicar la configuración que requieren los eventos a monitorizar (p.ej. con el flag `pmc` se indica que PMC se ha de usar para monitorizar un evento y el valor de la configuración que requiere dicho PMC). Los flags se mantienen en una lista distinta para cada arquitectura, con la nueva inclusión en ARM se corresponden a los siguientes:

```
static pmu_flag_t pmu_flags[]= {
    {"pmc",8,0},
    {"usr",1,0},
    {"os",1,0},
    {"ebs",32,0},
    {"pmu",8,0},
    {"systemwide",1,0},
    {"coretype",1,1},
    {NULL,0,0}
};
```

Los nuevos flags incluidos, en el caso de ARM, son `pmu` y `systemwide`. El flag `pmu` ahora permite hacer uso de cualquiera tipo de PMU –junto con su configuración– en los eventos ofrecidos por perf_events en el sistema. El otro flag introducido, `systemwide`, permite monitorizar eventos de perf_events desde todas las CPUs, sin estar limitado a una tarea o *core* específico. Ambos flags permiten la configuración de la gama completa de eventos *hardware* y *software* de perf_events disponibles en arquitecturas ARM cuando se inicialicen las PMUs.

También se han modificado ligeramente los mecanismos de construcción de la estructura `pmu_props_cpu` –información sobre las propiedades de una PMU– para que incluyan soporte a ARM32 y ARM64 simultáneamente. Esto se consigue haciendo uso de la nueva tabla de procesadores ARM soportados para hacer la correspondencia. Se hace una correspondencia entre el identificador adquirido del *hardware* y el nombre que se le ha dado en PMCTrack en la tabla.

En arquitecturas **x86**, para ampliar los eventos disponibles a usar en el backend se han incluido también nuevos flags de configuración el la lista de flags disponibles.

```
static pmu_flag_t pmu_flags[]= {
    {"pmc",8,0},
    [...]
    {"systemwide",1,0},
    {"config1",64,0},
    {"config2",64,0},
    {"pmu",8,0},
    {NULL,0,0}
};
```

Al igual que en arquitecturas ARM, entre los nuevos flags introducidos se encuentran también los flags `pmu` y `systemwide`, para dar soporte a cualquier tipo de PMU de los eventos disponibles en perf_events.

En x86 los contadores de monitorización del rendimiento son más complejos, esto hace que existan sub-eventos en perf_events (véase sección 3.5). El acceso a estos sub-eventos se realiza mediante la combinación de máscaras de selección. Para dar soporte a estos sub-eventos y eventos que necesiten más de 64 bits para su configuración, se han introducido dos nuevos flags para x86 en el backend, `config1` y `config2`. La codificación de estos flags depende de cada evento; no todos los eventos necesitan de estos flags para su configuración.

La inclusión de estos cuatro nuevos flags en x86 dan soporte en PMCTrack a toda la colección de eventos de perf_events *hardware* y *software* disponibles, cuando se inicialicen las PMUs.

En cuanto a los mecanismos presentes para averiguar el modelo y propiedades del procesador y las PMUs existentes junto con sus propiedades, se han implementado ligeras modificaciones sobre el código ya presente con soporte de Intel para incluir también el soporte de descubrimiento sobre procesadores AMD de familias mencionadas anteriormente.

Tras todas las modificaciones realizadas, ahora la entrada */proc/pmc/info* muestra la inclusión de los nuevos flags de PMUs para el nuevo soporte de eventos de perf_events más allá de los asociados a contadores *hardware* de rendimiento. Como se expone a continuación, en arquitecturas x86 ahora se muestran los nuevos flags incluidos (`pmu`, `config1`, `config2` y `systemwide`):

```
$ cat /proc/pmc/info
*** PMU Info ***
nr_core_types=1
[PMU coretype0]
pmu_model=x86_intel-core.broadwell
nr_gp_pmcs=8
nr_ff_pmcs=3
pmc_bitwidth=48
flags=[pmc(8),usr,os,umask(8),cmask(8),edge,inv,any,ebs(32),systemwide,coretype/g,
config1(64),config2(64),pmu(8)]
[...]
```

5.3.2 Análisis de la configuración de eventos

Cuando se hace uso de la herramienta de línea de comandos `pmctrack` se le indica a través de la opción `-c` qué eventos se desean monitorizar utilizando una cadena de configuración de PMCs, en formato *raw* o basada en mnemotécnicos. Para el caso concreto del backend de perf_events, esta cadena representa la configuración que se desea de cada evento perf_events a monitorizar.

PMCTrack guarda la correspondencia de los mnemotécnicos dentro de ficheros en el directorio */etc/events/* de sus fuentes. Las cadenas de configuración que le son pasadas a PMCTrack se transforman a formato *raw*. Este formato es una representación de

bajo nivel orientada a escribir en los registros *hardware* para facilitar el análisis. Esta representación esta definida en términos de los flags de configuración de las PMUs establecidos anteriormente en `init_pmu()` para cada CPU. Un ejemplo de fichero de mnemotécnicos en PMCTrack es el fichero *x86_intel.broadwell.csv* para un procesador Intel x86 con microarquitectura Broadwell:

```
event_name,subevent_name,event_code,description,flags
instr_retired_fixed,-,-,-,type=fixed;pmc=0,
unhalted_core_cycles_fixed,-,-,-,type=fixed;pmc=1,
unhalted_ref_cycles_fixed,-,-,-,type=fixed;pmc=2,
instr,-,-,-,type=fixed;pmc=0,
cycles,-,-,-,type=fixed;pmc=1,
unhalted_core_cycles,-,0x3c,-,-,
instr_retired,-,0xc0,-,-,
unhalted_ref_cycles,-,0x3c,-,umask=0x1,
llc_references,-,0x2e,-,umask=0x4f,
llc_references,prefetch,0x2e,-,umask=0x7f,
llc_misses,-,0x2e,-,umask=0x41,
llc_misses,prefetch,0x2e,-,umask=0x71,
branch_instr_retired,-,0xc4,-,-,
branch_misses_retired,-,0xc5,-,-,
l2_references,-,0x24,-,umask=0xff,
l2_misses,-,0x24,-,umask=0x3f,
l2_lines_in,-,0xf1,-,umask=0x07,
uncore_event0,-,0xb7,-,umask=0x01,
uncore_event1,-,0xbb,-,umask=0x01,
```

Es posible visualizar los mnemotécnicos disponibles para configurar la monitorización en PMCTrack a través de la herramientas de comandos `pmc-events` de PMCTrack con la opción `-L`. Con esta misma herramienta provista por PMCTrack, también es posible ver la representación de bajo nivel a la que son convertidas las cadenas de configuración. En el siguiente ejemplo se muestra el uso de `pmc-events` para obtener el formato *raw*:

```
$ pmc-events instr:ebs,cycles:os,llc_misses,branch_misses_retired,uncore_event0
pmc0,ebs0,pmc1,os1,pmc3=0x2e,umask3=0x41,pmc4=0xc5,pmc5=0xb7,umask5=0x01
```

Como se ha mencionado anteriormente, los backends han de implementar la función `parse_pmcs_strconfig()`. Esta función hace un análisis de la cadena de configuración en formato *raw* –dividiendo en *tokens* los flags de configuración– para rellenar la estructura `pmc_usrcfg_t` con la configuración solicitada para un evento. Estas estructuras serán posteriormente utilizadas para configurar los eventos en la inicialización de las PMUs. La función `parse_pmcs_strconfig()` es compartida dentro del backend por todas las arquitecturas.

Con la inclusión de los nuevos flags –para la configuración de las PMUs asociadas a los eventos– en arquitecturas ARM y x86 (`pmu`, `config1`, `config2` y `systemwide`), ha sido necesario añadir nuevos campos a la estructura `pmc_usrcfg_t` para representar las nuevas opciones de configuración solicitadas.

```
typedef struct {
[...]
#if defined (CONFIG_PMC_PERF)
    int cfg_pmu;
    unsigned char cfg_systemwide;
    uint64_t cfg_config1;
    uint64_t cfg_config2;
    unsigned char cfg_force_enable; /* For inheritance in MT apps */
#endif
} pmc_usrcfg_t;
```

- `cfg_pmu`, es el ID de la PMU del evento a monitorizar. Disponible en el fichero */sys/devices/<event_source>/type* del sistema.
- `cfg_systemwide`, indica si el evento es configurado como *system-wide* o no.
- `cfg_config1` y `cfg_config2`, se utilizan para configurar eventos o sub-eventos que necesitan un registro adicional o que no caben en el campo de configuración normal.

Además de estos nuevos campos para los nuevos flags de configuración, el backend de perf_events hace uso del resto de campos que ya estaban definidos en `pmc_usrcfg_t`. El uso del resto campos se introdujo en la primera versión del backend y se usan para almacenar la configuración del resto de flags `pmc`, `ebs`, `os`, … Estos flags permiten utilizar los mnemotécnicos de PMCTrack para indicar la configuración a eventos perf_events asociados a contadores *hardware* de rendimiento.

También ha sido necesario añadir dentro de la función `parse_pmcs_strconfig()` nuevos análisis para los nuevos flags y su asignación en los nuevos campos de `pmc_usrcfg_t`. De esta forma se ha conseguido un nuevo soporte en PMCTrack a todos los tipos de eventos de perf_events (con el flag `pmu`), y sus posibles configuraciones (con los flags `config1`, `config2` y `sistemwide`) en la inicialización de dichos eventos.

Tras la implementación de estas modificaciones dentro del *parser* del backend de perf_events ahora PMCTrack está en disposición de hacer uso de la información que perf_events expone en el espacio de usuario a través de sysfs o la herramienta `perf`.

Cogiendo como ejemplo las PMUs relacionadas con el consumo de energía, como se ha visto en la sección 5.1 es posible obtener el tipo de `pmu` y la codificación hexadecimal del evento de la siguiente manera:

```
$ cat /sys/devices/power/type
10
$ cat /sys/devices/power/events/energy-pkg
event=0x02
$ cat /sys/devices/power/events/energy-ram
event=0x03
```

Una vez obtenidos el número entero que identifica a la PMU y la codificación de los eventos, se puede realizar una cadena de configuración en formato *raw* para PMCTrack.

```
$ pmc-events instr,cycles,0x2:pmu=10:systemwide,0x3:pmu=10:systemwide
pmc0,pmc1,pmc3=0x2,pmu3=10,systemwide3,pmc4=0x3,pmu4=10,systemwide4
```

Si por el contrario se desea obtener la información relacionada a un evento desde la herramienta `perf` con el comando `perf list --details` el procedimiento es similar. Tomando como ejemplo el evento `load_latency_gt_16`:

```
$ perf list --details
[...]
  mem_trans_retired.load_latency_gt_16
       [Loads with latency value being above 16 Spec update: BDM100, BDM35]
        cpu/umask=0x1,(null)=0x4e2b,event=0xcd,ldlat=0x10/
```

Se observa que el evento hace uso de la configuración `ldlat` dentro de perf_events. Como se explica en la sección 5.1, el formato de esta configuración se muestra en los ficheros *format* a través de sysfs, donde se indica que para este caso se usa el campo `config1`:

```
$ cat /sys/devices/cpu/format/ldlat
config1:0-15
```

Con toda la información necesaria ya a disposición, la construcción de la cadena de configuración para PMCTrack para hacer monitorización sobre este evento quedaría con el siguiente formato:

```
$ pmc-events instr,cycles,0xcd:umask=0x1:config1=0x10
pmc0,pmc1,pmc3=0xcd,umask3=0x1,config13=0x10
```

5.3.3 Configuración e inicialización de eventos a monitorizar

Por último ha sido necesario hacer cambios en la configuración e inicialización de los eventos a monitorizar. De esto se encarga la función `do_setup_pmcs()` de cada backend, la cual es dependiente de la arquitectura sobre la que se trabaje. Por esta razón se ha creado una nueva versión de la función `do_setup_pmcs()` dentro del backend para arquitecturas ARM y se ha modificado la implementación ya existente para Intel x86 de forma que tenga soporte también para AMD.

Esta función recibe la lista de estructuras `pmc_usrcfg_t` (rellenada por el *parser* del mismo backend) con una posición por cada evento con la configuración para dicho evento. También recibe una máscara con las posiciones a usar de esta lista, ya que no siempre son usados todos los contadores de rendimiento disponibles. En base a la configuración recibida, `do_setup_pmcs()` tiene que encargarse de crear para cada evento a monitorizar una máscara de bits con la configuración necesaria para los registros de configuración de los contadores de rendimiento.

Los registros de configuración se componen de campos como *enable*, *umask*, etc. Estos campos varían según el modelo de procesador y la ubicación de cada bit sólo se expone en los manuales de cada fabricante. Lo ideal para este backend es que perf_events ofreciera

en espacio del kernel mecanismos para conocer el *layout* de los registros de configuración o mecanismos –específicos de la arquitectura– para poder asignar correctamente los bits de los registro de configuración con la configuración deseada. Desafortunadamente perf_events no dispone de ningún mecanismo de este tipo en el espacio del kernel. Aunque perf_events ofrece un campo en la configuración asociada a un evento para indicarle la configuración que se introducirá dentro del registro de configuración *hardware*, no tiene ningún mecanismo que exponga los campos que componen los registros de configuración en espacio del kernel. Por ello crear una máscara de bits para los registros de configuración no es posible a través de perf_events.

En la versión de este backend que se introdujo en PMCTrack v1.5, dado que sólo tiene soporte para Intel y una colección de eventos perf_events limitada, la creación de esta máscara de bit no presentaba gran complejidad.

```
config_mask=(pmc_cfg[i].cfg_umask << 8 | pmc_cfg[i].cfg_evtsel);
```

Con la nueva introducción de soporte para otras arquitecturas y configuraciones más complejas, para soportar la colección de eventos de perf_events, la creación de la máscara de bits de los registros de configuración se complica bastante.

Para el resto de backends de PMCTrack, que no son el de perf_events, se creó un código para exponer una estructura de C en la que se ven representados los campos de los registros de configuración de cada arquitectura. A partir de esta estructura se crea un valor hexadecimal con los bits configurados para el registro de configuración. De esta forma el *layout* de los registros se exporta como una estructura de C y no es necesario realizar máscaras ni desplazamientos de bits de forma directa. Este mecanismo para crear el valor de configuración de los registros de configuración de las PMUs se encuentra en los ficheros `pmc_bit_layout.h` de las fuentes de PMCTrack y son dependientes de la arquitectura.

A causa de la limitación de perf_events y la necesidad de dar soporte a nuevas arquitectura en esta nueva versión del backend de perf_events, se ha introducido el uso de estos mecanismos de creación de la máscara de bits para los registros de configuración heredados del resto de backends de PMCTrack. Para ello se ha hecho la nueva inclusión de los ficheros `pmc_bit_layout.h` para ARM, Intel y AMD.

```
#include <pmc/arm64/pmc_bit_layout.h>
#include <pmc/arm/pmc_bit_layout.h>
#include <pmc/amd/pmc_bit_layout.h> /* Added Intel and AMD layout */
```

Haciendo uso de estos mecanismos ahora es posible crear la máscara de bits dentro de la estructura ofrecida para cada arquitectura. En comparación con la antigua creación de la máscara de bits para los registros de configuración en la arquitectura x86 Intel, ahora se tiene las siguiente implementación haciendo uso del código heredado de los `pmc_bit_layout.h`:

```
intel_evtsel_msr evtsel;
[...]
init_intel_evtsel_msr(&evtsel);
set_bit_field(&evtsel.m_evtsel, pmc_cfg[i].cfg_evtsel);
set_bit_field(&evtsel.m_umask, pmc_cfg[i].cfg_umask);
set_bit_field(&evtsel.m_e,pmc_cfg[i].cfg_edge);
set_bit_field(&evtsel.m_inv, pmc_cfg[i].cfg_inv);
set_bit_field(&evtsel.m_cmask, pmc_cfg[i].cfg_cmask);
set_bit_field(&evtsel.m_any, pmc_cfg[i].cfg_any);
config_mask=evtsel.m_value;
```

Tras la asignación de los distintos campos –mediante `set_bit_field`– se puede obtener la máscara de bits simplemente accediendo al campo `m_value`. Para arquitecturas ARM y AMD la creación de `config_mask` usando el código heredado de los otros backends es similar.

Una vez creada la máscara de bits, para establecer los campos de los registros de configuración, ya se está en disposición de inicializar los eventos a monitorizar. Dentro de `do_setup_pmcs()` se llama a la función `init_counter()`. La implementación de esta función es compartida por todas las arquitecturas dentro del backend y será la encargada de inicializar el evento perf_events con la configuración de su `pmc_usrcfg_t` y la máscara de bits para los registros de configuración creada.

La función `init_counter()` ha sufrido los cambios más relevantes frente a la versión disponible en PMCTrack v1.5. La creación de la configuración del evento de perf_events a monitorizar de la que se parte en la versión v1.5 es la siguiente:

```
struct perf_event_attr sched_perf_hw_attr = {
    .type           = PERF_TYPE_RAW, /* For consistency we always use RAW events */
    .size           = sizeof(struct perf_event_attr),
    .disabled       = 1, /* Critical: alwas disabled on creation */
    .exclude_kernel = (pmc_cfg->cfg_os==0 && pmc_cfg->cfg_usr==1),
};
sched_perf_hw_attr.config = config_mask;
```

En la versión de partida sólo es posible crear eventos asociados a contadores *hardware*, el tipo de PMU es siempre `PERF_TYPE_RAW`. Con este tipo de PMU, es necesario proporcionar la codificación hexadecimal (directa) *raw* para el registro de configuración del contador de rendimiento en el campo `config`. El tipo `PERF_TYPE_RAW` hace que los eventos perf_events que no provienen de contadores *hardware* (aquellos que provienen de PMUs del kernel, consumo energético, fallos de página, cambios de contexto, etc) no puedan ser codificados por esta PMU [78].

Con la inclusión de los nuevos flags, la creación de la configuración del evento de perf_events a monitorizar a pasado a ser la siguiente:

```
struct perf_event_attr perf_hw_attr = {
    .type           = pmc_cfg->cfg_pmu!=-1? pmc_cfg->cfg_pmu : PERF_TYPE_RAW,
    .size           = sizeof(struct perf_event_attr),
```

```
    .disabled       = !pmc_cfg->cfg_force_enable,
    .exclude_kernel = (pmc_cfg->cfg_os==0 && pmc_cfg->cfg_usr==1),
    .config1 = pmc_cfg->cfg_config1,
    .config2 = pmc_cfg->cfg_config2,
};
perf_hw_attr.config = config_mask;
```

Los atributos del evento a monitorizar se han modificado para que el tipo sea el propio flag nuevo `pmu`, pudiendo así seleccionar los distintos tipos de eventos que ofrece perf_events. A través de este flag ahora se le puede indicar el tipo de evento mediante la opción `-c <event>:pmu=<value>` de la herramienta de línea de comandos `pmctrack`. Ahora que también se ha introducido el nuevo soporte en el backend para los flags `config1` y `config2`, se está en disposición de poder establecer los campos `config1` y `config2` para el evento. De la misma forma, ahora se puede indicar nuevas configuraciones de eventos que antes no eran posibles mediante las opciones `-c config1=<value>,config2=<value>`.

El paso de la configuración del contador de rendimiento a los registros de configuración se realiza mediante la asignación de la nueva máscara de bits al campo `config` de la estructura de atributos asociada a cada evento. Este campo se tomará en conjunto con los campos `config1` y `config2` si son necesarios para configurar un evento específico.

Esto completa el soporte de PMCTrack a eventos que no sólo provienen de contadores *hardware* sino a todos aquellos eventos *software* y *hardware* de perf_events, incluso aquellos casos en los que 64 bits –del campo de configuración `config`– no son suficientes para especificar completamente el evento.

Para eventos configurados como *system-wide* –es decir, para los que se recolecta datos de todas las CPUs– con el nuevo flag `systemwide` es necesario realizar además una configuración especial para que funcione correctamente perf_events, como se muestra a continuación:

```
if (pmc_cfg->cfg_systemwide){
    perf_hw_attr.exclude_kernel=0;
    task=NULL;
    cpu=0;
}
```

Para que esta configuración funcione correctamente perf_events requiere que se especifique como `NULL` la tarea a la que se asocia el evento, indicar la CPU cero, y establecer que el evento no excluya el conteo de eventos que ocurren en el espacio del kernel. La necesidad de esta configuración –al igual que otras muchas configuraciones de perf_events– no está documentada por perf_events y ha tenido que ser estudiada a partir del código fuente de Linux.

Para eventos configurados como *system-wide* es necesario ser *root* para invocar a perf_events. Una alternativa para hacer uso de estos eventos desde un usuario no *root* es modificar el nivel de perf_events *paranoid* como *root*:

```
# sysctl kernel.perf_event_paranoid=-1
```

El proceso de creación del evento con los atributos definidos anteriormente es similar a aquel descrito en la sección 4.3.2. Para ello, se hace uso de la función `perf_event_create_kernel_counter()` proporcionada por la API de perf_events y en este caso sí se le especifica una función manejadora, que se ejecutará cada vez que se reciba la interrupción de desbordamiento de un evento. Cuando se alcanza el umbral de desbordamiento establecido se produce una interrupción que es capturada por perf_events. Ésta desencadenará la invocación de la función `event_overflow_callback()` definida en el backend de perf_events la cual ejecutará `do_count_on_overflow()` implementada por el núcleo independiente de la plataforma de PMCTrack. A continuación se muestra la codificación completa de la invocación a `perf_event_create_kernel_counter()` para crear el nuevo evento:

```
event = perf_event_create_kernel_counter(
    &perf_hw_attr,
    cpu, task,
    pmc_cfg->cfg_ebs_mode?event_overflow_callback:NULL
#if LINUX_VERSION_CODE > KERNEL_VERSION(3, 2, 0)
    , NULL
#endif
);
```

La implementación de la función de desbordamiento para conteos en modo EBS, `event_overflow_callback()`, se ha mantenido sin realizar ninguna modificación con respecto a la versión v1.5.

Completados todos los cambios que se han explicado, el backend de perf_events de PMCTrack ya se encuentra en un estado en el que ofrece nuevo soporte a arquitecturas ARM y AMD. Otro gran avance que se ha logrado con estas modificaciones es la posibilidad de hacer uso de la colección completa de eventos de perf_events disponibles en el sistema.

Continuando con el ejemplo expuesto en las secciones anteriores, tras las modificaciones realizadas sobre `do_setup_pmcs` y `init_counter` del backend, PMCTrack ahora es capaz de monitorizar los eventos de perf_events del consumo energético global o relacionado con la RAM vistos anteriormente. Como se mostraba en la sección 5.1, se ha de tener en cuenta las unidades y la escala en que perf_events reporta estos eventos:

```
$ cat /sys/devices/power/events/energy-pkg.unit
Joules
$ cat /sys/devices/power/events/energy-pkg.scale
2.3283064365386962890625e-10
```

Dado que PMCTrack muestra la energía en Watts es necesario hacer una conversión de las métricas obtenidas por la herramienta de línea de comandos `pmctrack`. PMCTrack ofrece la posibilidad de hacer la conversión mediante su herramienta de línea de comandos `pmc-metric`, la cual se puede usar durante la ejecución de `pmctrack` de la siguiente

forma:

```
$ pmctrack -N 2 -T 0.3 -E -c instr,cycles,0x2:pmu=10:systemwide,0x3:pmu=10:systemwide \
./benchmarks/common/gamess06 | pmc-metric \
-m 'power-pkg=(pmc3*(2.3283**(-10.0)))/etime_us' \
-m 'power-ram=(pmc4*(2.3283**(-10.0)))/etime_us'
nsample    pid      event    power-pkg    power-ram
      1  12522       tick    14.161230     1.871387
      2  12522       tick    15.766435     2.222044
      3  12522       tick    15.758275     2.095002
      4  12522       tick    15.702005     2.137110
      5  12522       tick    15.614203     2.103093
```

En la monitorización que se acaba de mostrar se ha escogido tanto el consumo energético global y el relacionado con la RAM. Para ello se ha hecho uso del tipo de `pmu` y la codificación hexadecimal del evento obtenidos a través del sysfs y se han configurado ambos eventos como *system-wide*.

Ahora también es posible la monitorización de eventos de *cores* de distinto tipo en arquitecturas big.LITTLE de ARM. El uso de estos *cores* se expone con distintos tipos de PMUs, uno por *core*, para configurar un evento.

```
$ ls /sys/bus/event_source/devices
armv7_cortex_a15  armv7_cortex_a7  breakpoint  software  tracepoint
$ cat /sys/bus/event_source/devices/armv7_cortex_a15/type
7
$ cat /sys/bus/event_source/devices/armv7_cortex_a7/type
6
```

Como ejemplo se muestra la monitorización de las instrucciones y ciclos de un procesador con arquitectura ARM32. La plataforma que se ha usado en concreto es una placa ODROID-XU4, con un procesador Exynos5422 con arquitectura big.LITTLE (Cortex-A15 y Cortex-A7) [79]. Los *cores* "*big*" (Cortex-A15) se corresponden a las CPUs de la 4 a la 7 y los *cores* "*LITTLE*" (Cortex-A7) corresponden a las CPUs de la 0 a la 3.

```
$ pmctrack -c instr:pmu=7,cycles:pmu=7 -b 4 -T 0.5 ./benchmarks/common/libquantum06
[Event-to-counter mappings]
pmc1=instr
pmc2=cycles
[Event counts]
nsample    pid      event          pmc1          pmc2
      1  28831       tick    1126606617     822526818
      2  28831       tick     336716897     959994934
      3  28831       tick     605095772    1010899112
      4  28831       tick     725769914    1011394420
      5  28831       tick     798462681    1011509999
```

```
$ pmctrack -c instr:pmu=6,cycles:pmu=6 -b 1 -T 0.5 ./benchmarks/common/libquantum06
[Event-to-counter mappings]
pmc1=instr
pmc2=cycles
[Event counts]
```

```
nsample    pid      event          pmc1          pmc2
      1  28838       tick     349516800     566784514
      2  28838       tick     448590927     757282169
      3  28838       tick     248567664     744454330
      4  28838       tick     160862834     713139829
      5  28838       tick     158581416     755076082
```

5.4 Adaptación de la nueva API del kernel de PMCTrack para dar soporte en ARM

Logrado el nuevo soporte del backend de perf_events de PMCTrack para arquitecturas ARM, aún era necesario probar la compatibilidad de la nueva implementación de la API de PMCTrack sobre ARM32 y ARM64.

Para adaptar la nueva implementación de la API del kernel de PMCTrack –resultado de la implementación expuesta en el capítulo 4– se hizo uso del nuevo backend de perf_events –fruto de la expansión explicada en las secciones anteriores–. A su vez se han utilizado las siguientes plataformas: ODROID-XU4 (presentada en la sección anterior) con un procesador con arquitectura AMR32 big.LITTLE, y la plataforma HiKey960, con un procesador con arquitectura AMR64 big.LITTLE (4 *cores* Cortex-A73 y 4 *cores* Cortex-A53) [80].

Tanto ftrace como tracepoints invocan a las callbacks instaladas con expropiación deshabilitada. Esto supone que cualquiera que sea la funcionalidad que se implementa dentro de la callback, ésta no puede ser bloqueante. Durante la puesta a prueba de la nueva implementación de la API de PMCTrack en arquitecturas ARM, la desactivación de la expropiación por parte de ftrace y tracepoints para algunas callbacks de PMCTrack, resultó en errores graves del kernel.

El primer problema que surgió fue la incompatibilidad de trazar con ftrace la función `security_task_alloc()` para instalar la callback **pmcs_alloc_per_thread_data**. Como ya se ha mencionado en la sección 4.3.1, esto hizo que se escogiera la siguiente función trazable por ftrace, `copy_semundo()`, para instalar esta callback de PMCTrack.

Tras el cambio de la función del kernel a trazar, el sistema sufría bloqueo (*hang up*) al ejecutar la monitorización con PMCTrack sobre una aplicación. Este bloqueo se reportaba mediante el siguiente *log* en ARM32 y uno similar en ARM64:

```
[154721.314075] BUG: scheduling while atomic: bash/1073/0x00000002
[154721.318562] Modules linked in: pmc_arm_module(O) fuse rfkill loop
cpufreq_conservative cpufreq_userspace cpufreq_]
[154721.341879] Preemption disabled at:
[154721.341893] [<c0156e40>] sched_fork+0x20/0x34c
[154721.349916] CPU: 5 PID: 1073 Comm: bash Tainted: G      O    4.14.180-vanilla #1
[154721.357762] Hardware name: ODROID-XU4
[154721.361499] [<c0111d0c>] (unwind_backtrace) from [<c010d4ac>] ...
```

```
[154721.369294] [<c010d4ac>] (show_stack) from [<c096badc>] ...
[154721.376574] [<c096badc>] (dump_stack) from [<c0152ce4>] ...
[154721.384197] [<c0152ce4>] (__schedule_bug) from [<c09811a4>] ...
[154721.391994] [<c09811a4>] (__schedule) from [<c098162c>] ...
[154721.399099] [<c098162c>] (schedule) from [<c09851e4>] ...
[154721.406898] [<c09851e4>] (schedule_timeout) from [<c0982440>] ...
[154721.415216] [<c0982440>] (wait_for_common) from [<c098250c>] ...
[154721.423710] [<c098250c>] (wait_for_completion) from [<c0192318>] ...
[154721.432202] [<c0192318>] (__wait_rcu_gp) from [<c01967d0>] ...
[154721.440352] [<c01967d0>] (synchronize_sched) from [<c0240d74>] ...
[154721.449619] [<c0240d74>] (perf_event_alloc.part.14) from [<c0240ef0>] ...
[154721.458630] [<c0240ef0>] (perf_event_alloc) from [<c0244980>] ...
[154721.468426] [<c0244980>] (perf_event_create_kernel_counter) from  ...
[154721.480566] [<bf27330c>] (create_pmctrack_task_event [pmc_arm_module]) ...
[154721.493035] [<bf2733f4>] (fh_ftrace_sched_fork [pmc_arm_module]) from ...
[154721.503605] [<c01f6df8>] (ftrace_ops_list_func) from [<c01108a8>] ...
[154721.522590] ------------[ cut here ]------------
```

Lo más importante a entender de este *log* es el fallo que ocurre al intentar planificar cuando se están realizando operaciones atómicas en `perf_event_create_kernel_counter()` –en el momento de crear un nuevo evento perf_events– con expropiación deshabilitada. Esta función, como muestra el *log*, se invoca desde la función de enganche `fh_ftrace_sched_fork()`, función que se ha instalado con ftrace para que ejecute la callback **pmcs_alloc_per_thread_data** de PMCTrack.

El momento en el que se crea un nuevo evento de perf_events en **pmcs_alloc_per_thread_data** –para inicializar los datos de monitorización por hilo para PMCTrack– es precisamente cuando se realiza la invocación a la función `perf_event_create_kernel_counter()` de la API de perf_events. Esta función hace uso de `perf_event_alloc()` que reserva memoria para el nuevo evento. Para ello hace una llamada a `kzalloc()` con el flag `GFP_KERNEL`. Realmente `kzalloc()` se encarga simplemente de invocar a `kmalloc()` con un flag específico para indicar que se rellene la memoria con ceros, además de los flags que se le son indicados por `kzalloc()`:

```
static inline void *kzalloc(size_t size, gfp_t flags)
{
    return kmalloc(size, flags | __GFP_ZERO);
}
```

Lo importante de esta llamada que hace perf_events es el flag `GFP_KERNEL` que se le indica a `kmalloc()`. Este flag indica que el flujo de ejecución que invoca la llamada puede ser expropiado durante la asignación de la memoria. Dado que la callback **pmcs_alloc_per_thread_data**, donde se realiza la creación del nuevo evento perf_events, es invocada por ftrace con expropiación deshabilitada, al permitir a `kmalloc()` ser expropiada al reservar memoria resulta en error y bloqueo completo del sistema en ARM.

Ftrace deshabilita la expropiación en su función "trampolín" `__ftrace_ops_list_func()`

(explicada en la sección 3.3.2.1) ejecutando la función `preempt_disable_notrace()`, y una vez realizada la ejecución de la callback se vuelve a habilitar con `preempt_enable_notrace()`.

La solución a este problema en la callback **pmcs_alloc_per_thread_data** consiste en volver a habilitar la expropiación dentro de la propia función de enganche instalada por ftrace.

```
static void notrace fh_ftrace_sched_fork(unsigned long ip, unsigned long parent_ip,
                                         struct ftrace_ops *ops, struct pt_regs *regs)
{
    struct task_struct* p=(struct task_struct*) regs_get_kernel_argument(regs, 1);
    preempt_enable_notrace();
    pmcs_alloc_per_thread_data(regs_get_kernel_argument(regs, 0), p);
    preempt_disable_notrace();
}
```

Solucionado el problema en la callback **pmcs_alloc_per_thread_data**, el uso de PMCTrack causaba el siguiente *log* de error resultado del bloqueo completo del sistema.

```
[12948.703498] WARNING: CPU: 5 PID: 4782 at kernel/smp.c:291 smp_call_function_single
[12948.712058] Modules linked in: mchw_perf(O) fuse rfkill loop cpufreq_conservative
cpufreq_userspace cpufreq_powers]
[12948.735287] CPU: 5 PID: 4782 Comm: bash Tainted: G       O    4.14.180-vanilla #1
[12948.743081] Hardware name: ODROID-XU4
[12948.746733] [<c0111d0c>] (unwind_backtrace) from [<c010d4ac>] ...
[12948.754441] [<c010d4ac>] (show_stack) from [<c096badc>] (dump_stack+0x8c/0xa0)
[12948.761633] [<c096badc>] (dump_stack) from [<c01278bc>] (__warn+0xf8/0x110)
[12948.768563] [<c01278bc>] (__warn) from [<c01279a4>] ...
[12948.776103] [<c01279a4>] (warn_slowpath_null) from [<c01b9fcc>] ...
[12948.785377] [<c01b9fcc>] (smp_call_function_single) from [<c023c23c>] ..
[12948.794302] [<c023c23c>] (perf_event_read) from [<c023c3e0>] ...
[12948.802970] [<c023c3e0>] (perf_event_read_value) from [<bf2238f8>] ...
[12948.813294] [<bf2238f8>] (do_count_mc_experiment [mchw_perf]) from ...
[12948.824198] [<bf223afc>] (mod_exit_thread [mchw_perf]) from ...
[12948.834427] [<bf229204>] (pmcs_exit_thread [mchw_perf]) from [<bf229230>] ...
[12948.844822] [<bf229230>] (fh_ftrace_do_exit [mchw_perf]) from [<c01f6df8>]...
[12948.854609] [<c01f6df8>] (ftrace_ops_list_func) from [<c01108a8>] ...
[12948.863619] ---[ end trace 3996ec6989ed4fee ]---
```

En este caso, el *log* alude a la callback **pmcs_exit_thread** instalada mediante tracepoints con la función de enganche `fh_ftrace_do_exit()`. Al igual que ftrace, tracepoints ejecuta las callbacks asociadas a un tracepoint con expropiación deshabilitada. Esto lo realiza haciendo también uso de la función `preempt_enable_notrace()` para deshabilitar la expropiación y una vez terminada la ejecución de la callback rehabilitándola con `preempt_disable_notrace()`.

El error surge con la invocación a la función `do_count_mc_experiment()` de PMCTrack. Ésta lee los contadores de rendimiento y actualiza los contadores en la estructura `pmon_prof_t` específica del hilo. En el backend de perf_events, la lectura de contado-

res se realiza con `perf_event_read_value()`, la cadena de llamadas de esta función acaba invocando a `smp_call_function_single()` la cual produce un bloqueo mediante `smp_cond_load_acquire()` [81] en la CPU sobre la que se hace la lectura de los contadores.

La solución a este problema en la callback **pmcs_exit_thread** es la habilitación de la expropiación dentro de la propia función de enganche instalada al tracepoint, como se muestra a continuación:

```
static void probe_sched_process_exit(void *data, struct task_struct *p)
{
    preempt_enable_notrace();
    pmcs_exit_thread(p);
    preempt_disable_notrace();
}
```

El último problema con la nueva implementación de la API de PMCTrack está presente sólo en arquitecturas ARM64. En este tipo de arquitecturas la posibilidad de tener ftrace configurado con el símbolo de preprocesado `DYNAMIC_FTRACE_WITH_REGS` –configuración que permite a ftrace hacer el trazado guardando el contexto para la callback instalada– no ha sido posible hasta la versión 5.5.19 de Linux [82], [83]. Esto se debe a que no existe soporte en ARM64 para `gcc -mfentry` [84], configuración del compilador `gcc` para que se inserte la llamada especial `__fentry__` en el prólogo de cada función del código fuente del kernel Linux (véase sección 3.3.2.1). Para solventar esto se ha creado una nueva opción `-fpatchable-function-entry` para `gcc` dando así soporte a ftrace con `DYNAMIC_FTRACE_WITH_REGS` [83], [85]. Esta nueva opción está sólo disponible a partir de la versión 8 de `gcc`. Todo esto implica que la nueva implementación de la API de PMCTrack en arquitecturas ARM64 sólo es compatible a partir de la versión 5.5.19 de Linux.

Tras la resolución de los problemas que presentaban el uso de ftrace y tracepoints con las callback de PMCTrack en ARM y teniendo en cuenta que en ARM64 el uso que se da a ftrace sólo es compatible a partir de la versión 5.5.19, la nueva implementación de la API de PMCTrack ha pasado a ser compatible con arquitecturas ARM.

5.5 Soporte de la nueva implementación de la API de PMCTrack con backends *legacy*

Otro de los objetivos tras completar la extensión de funcionalidad del backend de perf_events, era hacer compatible la implementación de la nueva API de PMCTrack con el resto de backends de PMCTrack, que no son el de perf_events. El único problema que se presenta al usar el resto de backends con la nueva implementación de la API de PMCTrack ocurre en la implementación del modo EBS (*Event-Based Sampling*) de PMCTrack para arquitecturas x86. Este modo de ejecución permite la recolección de los valores de monitorización de una aplicación cada vez que un evento alcanza un recuento

determinado.

En el resto de backends específicos de la arquitectura, cuando son ejecutados en modo EBS, PMCTrack puede capturar la interrupción de desbordamiento (*overflow*) de los contadores de monitorización del rendimiento, que se produce cuando estos llegan a un recuento indicado.

En x86 esta interrupción es una interrupción tipo NMI (*Non-Maskable Interrupt*). Cuando perf_events se encuentra activo en el sistema cuenta con la función `perf_event_nmi_handler()`, la cual sirve como operación de registro de la interrupción de desbordamiento de los contadores. El hecho de que perf_events capture la interrupción NMI emitida por los PMCs hace imposible que los backends (específicos de la arquitectura) de PMCTrack reciban la interrupción, impidiendo así que funcione el modo EBS dentro de PMCTrack.

En la implementación de la API de PMCTrack a través de un parche del kernel, este problema se soluciona modificando la funcionalidad de los eventos de perf_events que abstraen a los contadores *hardware*. Al hacer esto, se impide que perf_events capture la interrupción de desbordamiento de los contadores *hardware*. Esto hace que sólo funcionen los eventos *software* de perf_events.

Con la nueva implementación de la API de PMCTrack se pretende evitar depender de un parche para el kernel que modifique la funcionalidad de perf_events. La opción que se ha seguido para solucionar este problema de compatibilidad con el resto de backends es la creación y aplicación de un parche en vivo sobre el kernel. La idea reside en modificar la función `perf_event_nmi_handler()` en tiempo de ejecución para que no trate la interrupción de desbordamiento de los contadores *hardware*, de forma que se ignore retornando simplemente `NMI_UNKNOWN` (un flag del kernel para indicar que no se conoce que tipo de interrupción NMI se ha recibido).

Ftrace es capaz de alterar el estado de los registros después de salir de la callback que acopla al principio de una función del kernel trazada. En el caso concreto de arquitecturas x86, cambiando el registro `%ip` –dirección de memoria a la siguiente instrucción a ejecutar– se puede cambiar una función del kernel por una propia. En otras palabras, se puede forzar al procesador a hacer un salto incondicional de la función trazada con ftrace a otra función implementada por uno mismo.

La función `perf_event_nmi_handler()` se encuentra dentro de las funciones del kernel posibles a hacer trazado con ftrace. Esto plantea la posibilidad de "secuestrar" la función `perf_event_nmi_handler()` trazada y tener otra función a la que se llame en su lugar. Para conseguir esto se debe establecer ciertos flags en la estructura `ftrace_ops` asociada a la callback a instalar con ftrace. Uno de estos flags es `FTRACE_OPS_FL_SAVE_REGS` para obtener los registros del kernel dentro de la callback y poder modificarlos. El otro flag necesario es `FTRACE_OPS_FL_IPMODIFY`, presente desde la versión 3.19 de Linux. Este flag permite modificar el campo `ip` –dirección de retorno a la función que se está trazando– de la estructura `pt_regs` que le es provista al callback [66]. La descripción en

detalle de estos flags se encuentra en la sección 4.2.3.

Para crear una función que reemplace a otra a través de ftrace, es necesario que la nueva función se declare con la misma definición –mismos argumentos y tipo de retorno– que la función a la cual va a reemplazar. De esta forma la función que se ha creado para reemplazar a `perf_event_nmi_handler()` es la siguiente:

```
static int livepatch_perf_event_nmi_handler(unsigned int cmd, struct pt_regs *regs)
{
    return NMI_UNKNOWN;
}
```

Esta función simplemente retornará que no reconoce la interrupción de desbordamiento de los contadores *hardware* que ha recibido y permitirá a los backends de PMCTrack capturarla.

Creada la nueva función que va a reemplazar a `perf_event_nmi_handler()`, es necesario crear también la callback que se acoplará en prólogo de `perf_event_nmi_handler()`. Esta función debe modificar el registro `%ip` para que apunte a la nueva función.

```
static void notrace fh_ftrace_perf_event_nmi_handler(unsigned long ip,
                                                     unsigned long parent_ip,
                                                     struct ftrace_ops *ops,
                                                     struct pt_regs *regs)
{
    regs->ip = (unsigned long) livepatch_perf_event_nmi_handler;
}
```

De esta manera se consigue que cada vez que la función `perf_event_nmi_handler()` se invoque, ésta no sea ejecutada y se redirija la ejecución a su nueva implementación.

Linux ofrece por defecto un framework para crear y aplicar parches en vivo en el kernel, *Livepatching* [53], [86]. *Livepatching* funciona utilizando ftrace para redirigir las llamadas a funciones del kernel a las nuevas funciones parcheadas. Además, utiliza mecanismos de acoplamiento a la inserción y eliminación de módulos para parchear módulos cargables del kernel. *Livepatching* también cuenta con directorios en el sysfs para conocer qué parches se aplican y qué funciones modifican [53].

Este framework de *Livepatching* esta orientado a la aplicación de parches de seguridad en vivo en situaciones en las que no se quiere o puede reiniciar el sistema. Es por esto que su uso no resulta el adecuado para la aplicación que se pretende dar en la API del kernel de PMCTrack y se ha optado por la creación de la implementación propia para aplicar un parche en vivo sobre la función `perf_event_nmi_handler()` mediante el uso de ftrace. Con la aplicación de este nuevo parche en vivo del kernel, se logra hacer compatible a la nueva implementación de la API de PMCTrack con el resto de backends de PMCTrack.

Capítulo 6

Evaluación experimental

En este capítulo se expone la evaluación realizada sobre la nueva implementación de la API del kernel de PMCTrack para detectar una posible sobrecarga en el rendimiento respecto a la presente en PMCTrack v1.5. Para ello se discuten las métricas de rendimiento obtenidas al lanzar los *benchmarks* de las suites SPEC CPU 2006 y 2017 mientras son monitorizados por PMCTrack, tanto con la nueva versión de la API de PMCTrack como con la versión de partida (v1.5). Se expone también la sobrecarga asociada a la nueva implementación de la interfaz de callbacks de la API de PMCTrack.

El capítulo se estructura en las siguientes secciones. En la **sección 6.1** se presentan las suites SPEC CPU y el tipo de monitorización que se ha realizado durante la recogida de datos para la evaluación experimental. En esta sección se describe también el sistema en el que se ha ejecutado la monitorización de los *benchmarks*. En la **sección 6.2** se analiza la posible sobrecarga en los tiempos de ejecución obtenidos al realizar la monitorización del rendimiento sobre ambos conjuntos de *benchmarks* de SPEC CPU. En la **sección 6.3** se realiza la evaluación de las métricas escogidas resultantes de la monitorización con PMCTrack en modo EBS de los *benchmarks* de ambas suites SPEC CPU. En la **sección 6.4** se hace la misma evaluación realizada en el modo EBS, pero esta vez haciendo la monitorización con PMCTrack en modo TBS. En la **sección 6.5** se estudia la posible sobrecarga que se puede haber introducido en la nueva versión de la API de PMCTrack al hacer uso de un evento perf_events para almacenar los datos de monitorización por hilo.

6.1 Descripción del entorno experimental

Con el fin de conseguir medir la sobrecarga se ha monitorizado con PMCTrack la ejecución de los *benchmarks* SPEC CPU de las *suites* CPU2006 y CPU2017, tanto con la nueva implementación de la API de PMCTrack como con la versión presente en PMCTrack v1.5. A partir de este momento se referirá a la nueva implementación de la

API de PMCTrack como PMCTrack v2.0. Dado que el objetivo es evaluar la nueva API del kernel de PMCTrack, para ambas versiones de PMCTrack (v2.0 y v1.5), se ha usado siempre para llevar a cabo las mediciones del rendimiento el backend de perf_events resultado de las expansiones descritas en el capítulo 5.

Los SPEC CPU conforman un conjunto de *benchmarks*[1] para comparar las diferencias del rendimiento entre CPUs y plataformas. Estos *benchmarks* han sido desarrollados a partir de aplicaciones del mundo real para permitir medir y comparar el rendimiento, estresando el procesador, el subsistema de memoria y la efectividad del compilador de un sistema [87]. Muchas empresas utilizan SPEC CPU como punto de partida para determinar un nivel mínimo de rendimiento base [88].

Para la evaluación experimental se han creado una serie de *scripts* que lanzan respectivamente los *benchmarks* de los SPEC CPU de 2017 y 2006 mientras su rendimiento se monitoriza por la herramienta de línea de comandos `pmctrack`. PMCTrack se puede ejecutar para la monitorización de una aplicación con dos modos de muestreo distintos:

- El modo **EBS** (*Event-Based Sampling*) se basa en eventos. Se recogen los valores de los PMCs y de los contadores virtuales en la monitorización de una aplicación cada vez que un evento –en los PMCs o en los contadores virtuales– alcanza un valor determinado.
- El modo **TBS** (*Time-Based Sampling*) está basado en tiempo. Se recogen a intervalos de tiempo regulares los valores de los PMCs y de los contadores virtuales para una determinada aplicación.

En los *scripts* creados se especifican los parámetros específicos que se le quieren pasar a la herramienta de línea de comandos `pmctrack` para la monitorización. La configuración de `pmctrack` dentro de los scripts varía según sea un tipo de monitorización en modo EBS o TBS. De manera general para todos los *benchmarks* ejecutados se ha fijado la CPU número 2 con la opción `-b` de `pmctrack`, y se han recogido los siguientes eventos:

- Número de instrucciones.
- Número de ciclos.
- Número de instrucciones de salto retiradas que fueron mal predichas por el procesador [89]. Se corresponde al mnémotecnico `branch_misses_retired` en PMCTrack.
- Número de instrucciones de salto retiradas. Su mnémotecnico en PMCTrack es `branch_instr_retired`.
- Consumo energético global[2], evento `energy-pkg`.
- Consumo energético de la memoria RAM (*Random-Access Memory*), evento `energy-ram`.

[1] El SPEC CPU2006 se compone de 29 *benchmarks* mientras que el SPEC CPU2017 se compone de 23 *benchmarks*.

[2] La monitorización de eventos *software* es posible tanto para PMCTrack v1.5 como v2.0 porque para la evaluación se hace uso de la nueva implementación del backend de perf_events.

- Número de instrucciones retiradas. Se corresponde al mnémotecnico `instr_retired` en PMCTrack.
- Número de ciclos del *core* cuando éste no se detiene. Tiene como mnémotecnico `unhalted_core_cycles` en PMCTrack.

Las configuraciones específicas que se le han indicado a `pmctrack` para el modo EBS y TBS respectivamente se explican en las siguientes secciones.

El lanzamiento de los *scripts* creados se ha realizado sobre una máquina que dispone de las siguientes especificaciones. Por procesador dispone del modelo Intel® Xeon® D-1540 con 8 *cores* 16 *threads* con una frecuencia base de 2.00GHz y con una caché compartida de último nivel de 12MBs. La plataforma dispone de 16GBs de memoria RAM. Esta máquina tiene un sistema operativo Debian GNU/Linux 10 (*buster*) en el que se encuentran instaladas múltiples versiones del kernel Linux. En particular se han hecho experimentos con las versiones del kernel Linux 5.4.35 con el parche de PMCTrack –parche publicado con la versión v1.5 y disponible en el repositorio oficial de PMCTrack– y 5.4.35 *vanilla*, sin ninguna modificación.

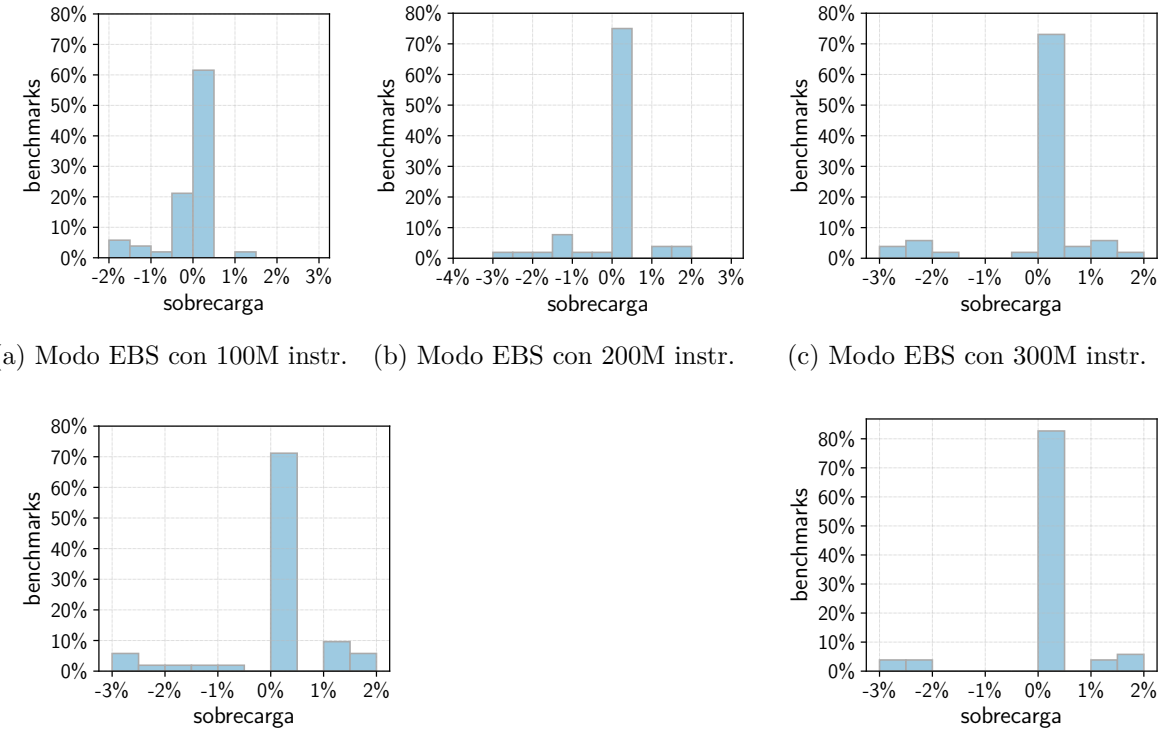
6.2 Tiempos de ejecución de SPEC CPU

Con el propósito de detectar una posible sobrecarga en los tiempos de ejecución de las dos colecciones de *benchmarks*, se ha realizado el lanzamiento de los 52 programas de SPEC CPU[3] con una serie de condiciones. La ejecución de PMCTrack se ha realizado en modo EBS y se ha especificado como evento –para recoger los valores de los PMCs y de los contadores virtuales– el alcance de cierto número de instrucciones. La ejecución del *benchmark* a través de PMCTrack se ha especificado para que finalice cuando se alcanza el número de muestras especificadas. Estas combinaciones del número de instrucciones y muestras que se han establecido para la ejecución de cada *benchmark* han sido las siguientes:

- 100 millones de instrucciones y 1500 muestras
- 200 millones de instrucciones y 750 muestras
- 300 millones de instrucciones y 500 muestras
- 400 millones de instrucciones y 375 muestras
- 500 millones de instrucciones y 300 muestras

Se puede observar que en la ejecución de cada una de estas condiciones siempre va a resultar en la ejecución de 150000 millones de instrucciones. Para evaluar la sobrecarga se han creado una serie de gráficas de distribución que se muestran en la figura 6.1. En esta figura (6.1), se muestra el porcentaje de sobrecarga en los tiempos de ejecución –de cada configuración EBS mostrada anteriormente– para la nueva implementación de la API del kernel de PMCTrack (v2.0) en relación a los tiempos de ejecución obtenidos para la versión presente en PMCTrack v1.5.

[3] 29 *benchmarks* del SPEC CPU2006 y 23 *benchmarks* del SPEC CPU2017.



(a) Modo EBS con 100M instr. (b) Modo EBS con 200M instr. (c) Modo EBS con 300M instr.

(d) Modo EBS con 400M instr. (e) Modo EBS con 500M instr.

Figura 6.1: Evaluación de la sobrecarga en los tiempos de ejecución para los SPEC CPU2006 y CPU2017

En las gráficas de distribución se muestra una sobrecarga siempre menor al 2% para un porcentaje muy reducido de *benchmarks*. Los *benchmarks* que presentan una sobrecarga no son los mismos para las distintas combinaciones del modo EBS monitorizadas. Por ejemplo, para el *benchmark* bzip206 con EBS configurado con 400M de instrucciones tiene un 1.4% de sobrecarga, configurado con 500M de instrucciones muestra un 2.5% de mejora (porcentaje de sobrecarga negativo), mientras que para el resto de monitorizaciones tiene un 0% de sobrecarga. Esto mismo sucede para el resto de *benchmarks* que muestran una variabilidad, por lo que no existe una correlación de la variación entre *benchmarks*.

De la misma forma, las gráficas de distribución muestran menor sobrecarga que v1.5 (valor negativo) de hasta un 3%. Esto se corresponde a una mejora del tiempo de ejecución en la nueva versión de la API del kernel de PMCTrack con respecto a la API de PMCTrack v1.5. Al igual que en el caso anterior esta variabilidad tampoco establece una correlación de la variación entre *benchmarks*, debido a que no son *benchmarks* específicos los que causan estas variaciones.

Esta variabilidad mostrada en el tiempo en un rango del 1% y 2% en favor de ambas versiones se puede considerar dentro del rango normal de variabilidad que se espera en

un sistema real.

Aun con la presencia de esta pequeña variabilidad normal, las gráficas de distribución revelan que no existe sobrecarga alguna (valor 0%) para un porcentaje muy elevado de *benchmarks*. Todo ello hace ver que la nueva implementación de la API de PMCTrack no introduce sobrecarga en relación a la API presente en PMCTrack v1.5.

6.3 Evaluación de las métricas de monitorización en modo EBS

Para el modo EBS se ha monitorizado la ejecución de los 23 *benchmarks* que componen el SPEC CPU 2017 y los 29 *benchmarks* del SPEC CPU 2006. Se ha realizado la recogida de muestreo con las 5 combinaciones de número de instrucciones y muestras en las opciones de monitorización de PMCTrack usadas en la sección anterior.

A partir de todos los eventos de monitorización que se han recogido con monitorización en modo EBS se han calculado múltiples métricas. De estas métricas, se han escogido para realizar la evaluación las siguientes: el IPC (instrucciones por ciclo) y el BRMPKINSTR (predicciones de salto fallidas por cada mil instrucciones).

Una primera evaluación corresponde a la realización de gráficas con la media aritmética de cada métrica escogida y su posterior media aritmética entre las 5 combinaciones de monitorización realizadas sobre cada *benchmark*.

La figura 6.2 muestra la comparación entre las medias de los IPCs de cada *benchmark* del SPEC CPU2017 y CPU2006 ejecutados sobre la nueva API de PMCTrack v2.0 y la API de PMCTrack v1.5. La gráfica muestra claramente un resultado estable y prácticamente similar entre las dos versiones de PMCTrack.

Durante la evaluación experimental se han realizado múltiples lanzamientos de los mismos *benchmarks* sobre el mismo sistema con las mismas opciones de monitorización de PMCTrack. Los resultados de monitorización siempre han presentado una variabilidad mínima entre distintas ejecuciones, aún tratándose con las mismas circunstancias sobre la misma versión de PMCTrack. Esta variabilidad mínima se corresponde con la reportada en la sección anterior (6.2) y se ha de ignorar aunque dé como resultado casos en los que se obtiene una insignificante mejora en las métricas para alguna de las versiones de la API de PMCTrack. Para la nueva implementación de la API de PMCTrack v2.0 algunos de los *benchmarks* donde se aprecia esta pequeña mejora son imagick17 (0.6%), povray06 (0.5%) y wrf06 (0.9%). Por el contrario también existen casos en los que la versión v1.5 muestra un IPC mínimamente mejor, estos casos se ven en los *benchmarks* deepsjeng17 (0.5%), gromacs06 (0.4%) y libquantum06 (0.2%).

De la misma forma que en el IPC, para el SPEC CPU 2017 y SPEC CPU 2006 se ha recopilado la comparación entre las medias aritméticas del BRMPKINSTR, de ambas versiones de la API de PMCTrack; esto se muestra en la figura 6.3. Esta gráfica muestra

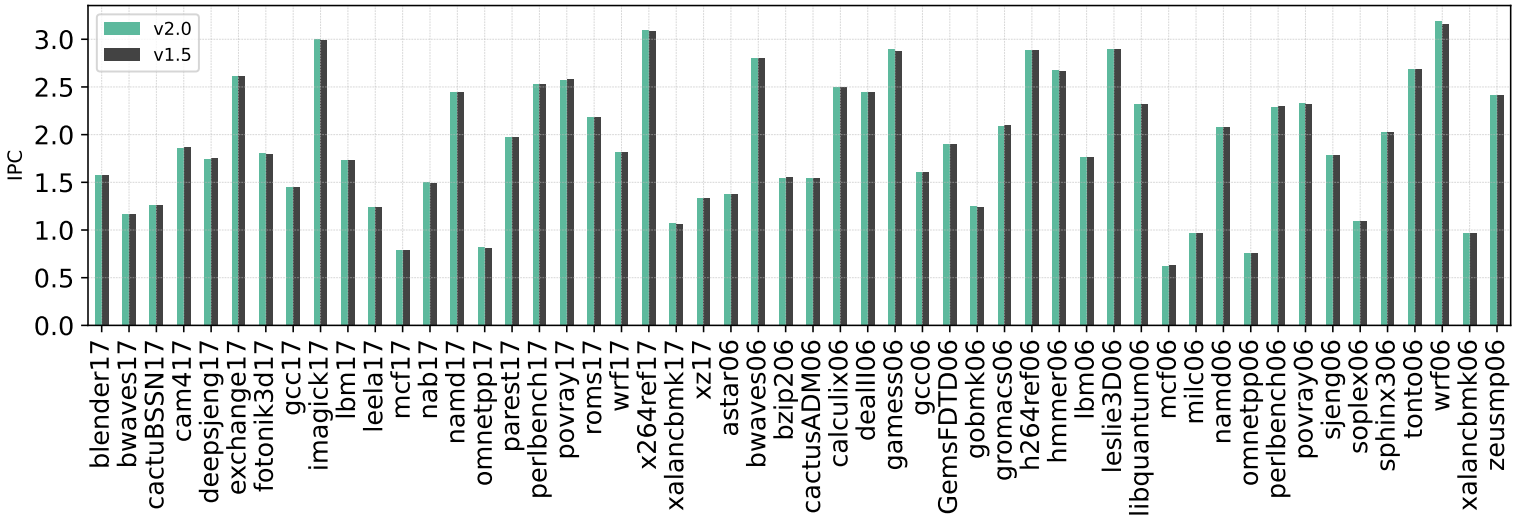


Figura 6.2: Media aritmética del IPC en modo EBS para los *benchmarks* de las suites SPEC CPU2006 y SPEC CPU2017

un resultado similar entre las dos versiones de PMCTrack. Sólo en el *benchmark* gobmk06 se puede apreciar una variación notable de un 1.1% en favor de PMCTrack v2.0, siendo un caso particular se puede atribuir a las variaciones comentadas anteriormente.

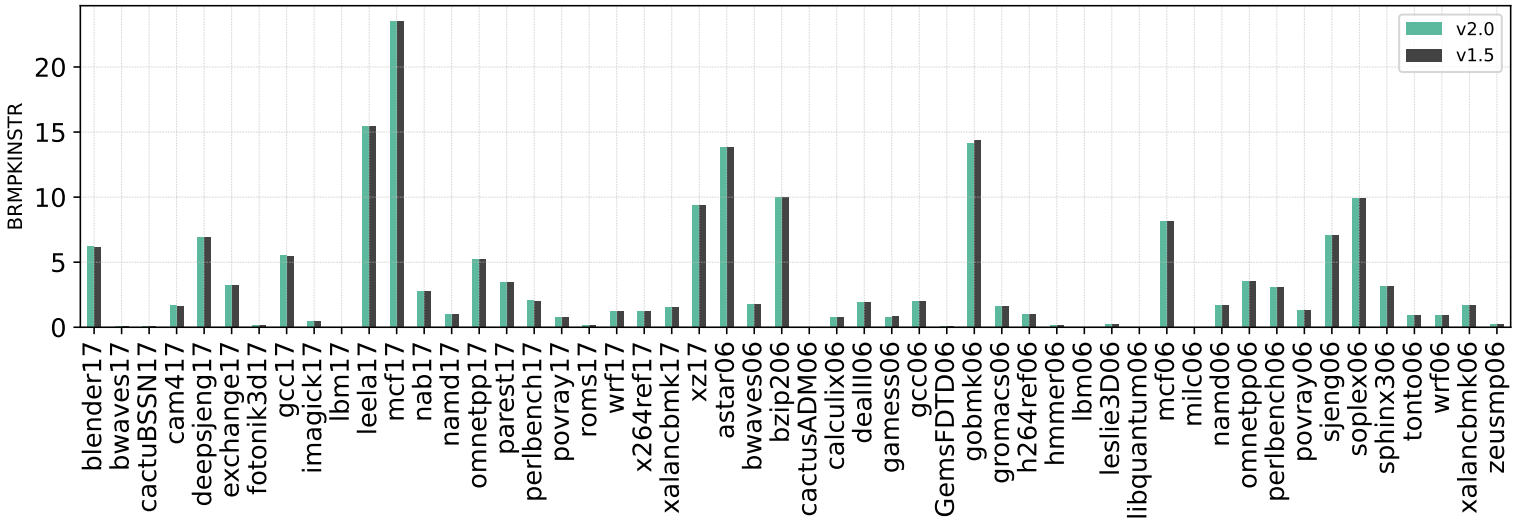
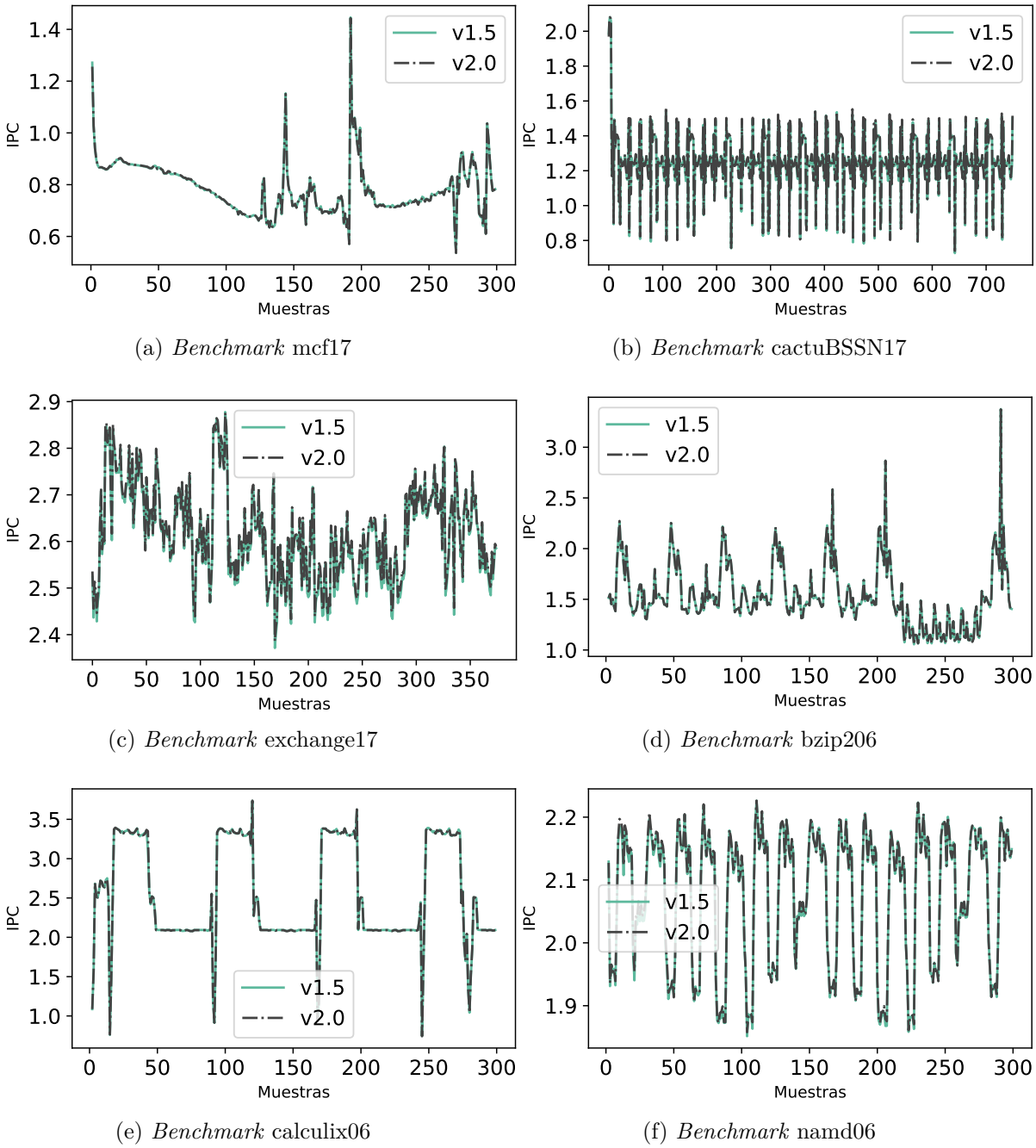


Figura 6.3: Media aritmética del BRMPKINSTR en modo EBS para ambas suites SPEC CPU2006 y SPEC CPU2017

Para poder analizar las métricas de *benchmarks* específicos se ha escogido un conjunto de *benchmarks* del SPEC CPU2017 y SPEC CPU2006 con distintas áreas de aplicación. En las figuras 6.4 y 6.5 se muestran las gráficas con el IPC y el BRMPKINSTR

respectivamente, obtenidos para cada *benchmark* de este conjunto. En todas estas gráficas la conclusión es la misma, las variaciones son mínimas entre las dos versiones. Esto demuestra que no se pierde rendimiento ni precisión en los datos recogidos durante la monitorización con PMCTrack v2.0.



(a) *Benchmark* mcf17

(b) *Benchmark* cactuBSSN17

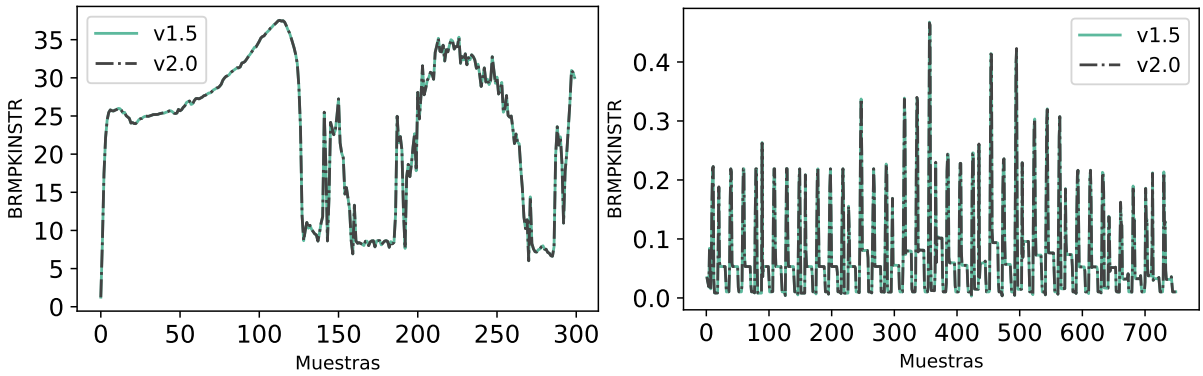
(c) *Benchmark* exchange17

(d) *Benchmark* bzip206

(e) *Benchmark* calculix06

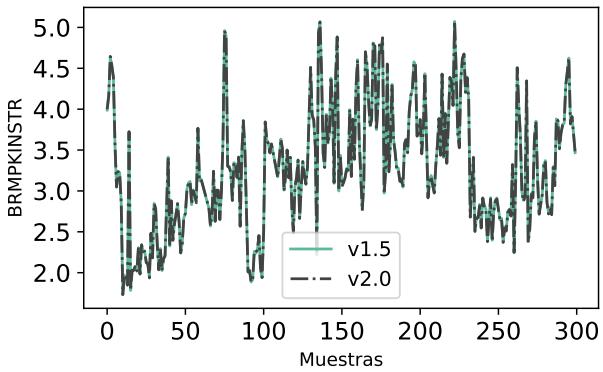
(f) *Benchmark* namd06

Figura 6.4: Evaluación del IPC en modo EBS

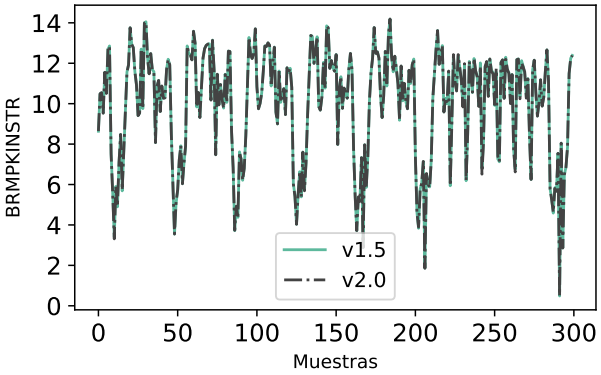


(a) *Benchmark* mcf17

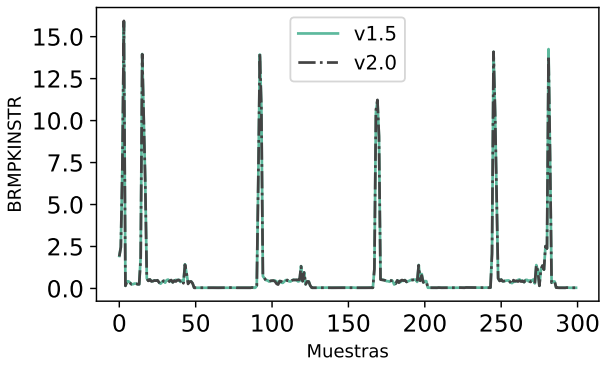
(b) *Benchmark* cactuBSSN17



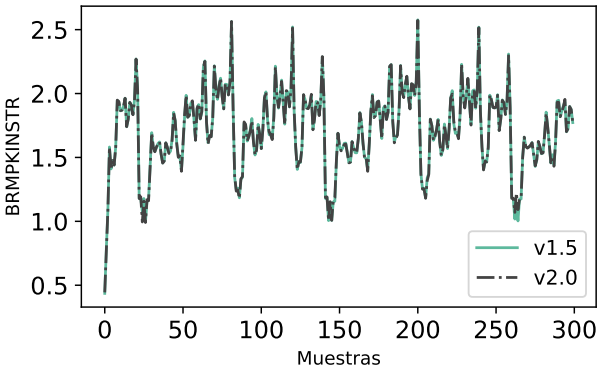
(c) *Benchmark* exchange17



(d) *Benchmark* bzip206



(e) *Benchmark* calculix06



(f) *Benchmark* namd06

Figura 6.5: Evaluación del BRMPKINSTR en modo EBS

6.4 Evaluación de las métricas de monitorización en modo TBS

La evaluación de la monitorización en modo TBS se ha realizado también sobre todos los *benchmarks* que componen el SPEC CPU2017 y el SPEC CPU2006. Al igual que en modo EBS, entre todas las métricas de monitorización que se han realizado para la evaluación –con los eventos monitorizados– se han elegido el número de instrucciones por ciclo (IPC) y el número de predicciones de salto fallidas por cada mil instrucciones (BRMPKINSTR) para la evaluación.

En este caso la monitorización con PMCTrack se ha configurado en los *scripts* en modo TBS para que el tiempo transcurrido entre dos muestreos consecutivos de los PMCs itere entre 100, 200, 300, 400 y 500 milisegundos para la ejecución de cada *benchmark*. De esta forma se han obtenido el resultado de cinco monitorizaciones para cada *benchmark* cada una con un intervalo de muestreo distinto.

La primera evaluación que se ha realizado es la representación en gráficos de la media aritmética de cada métrica escogida y su posterior media aritmética entre las 5 configuraciones de modo TBS sobre cada *benchmark*.

La figura 6.6 expone las medias de los IPCs de cada *benchmark* del SPEC CPU2017 y SPEC CPU2006. En este caso los resultados obtenidos para el IPC son consecuentes con los observados anteriormente en el modo EBS. Se tiene una pequeña variabilidad de un rango inferior al 1% en algunos de los *benchmarks*, mientras que el resto muestran un resultado prácticamente similar para ambas versiones de la API del kernel de PMCTrack.

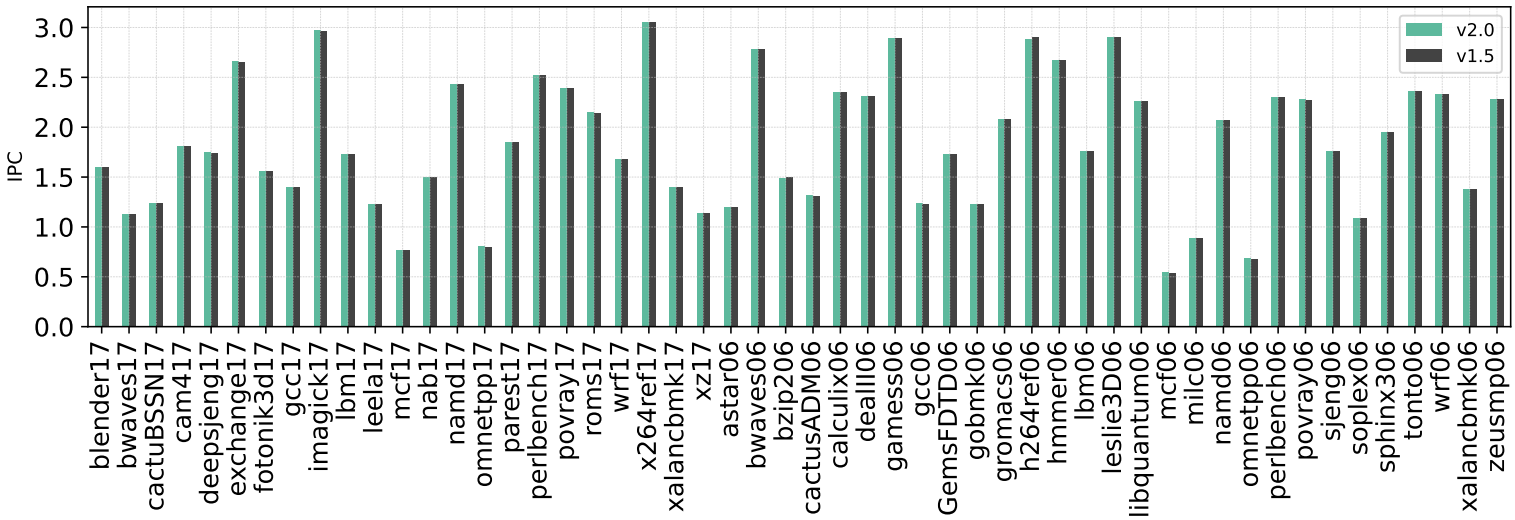


Figura 6.6: Media aritmética del IPC en modo TBS para los *benchmarks* de las suites SPEC CPU2006 y SPEC CPU2017

Algunos casos particulares de la figura 6.6 en los que se ve esta variación son imagick17,

que muestra una mejora del IPC del 0.4% en PMCTrack v2.0, y h264ref06, que por el contrario refleja un IPC 0.4% mejor en v1.5.

De la misma forma, para ambas suites SPEC CPU se ha recopilado la comparación entre las medias aritméticas del BRMPKINSTR. Estos resultados se muestran en la figura 6.7, en esta gráfica apenas existe una pequeña variabilidad de un rango del 1% en un conjunto muy reducido de los *benchmark*. Uno de los *benchmarks* donde se aprecia una pequeña mejora en PMCTrack v1.5 es deepsjeng17 (0.7%). Por el lado contrario, un *benchmark* que muestra una mejora en el BRMPKINSTR en la API de PMCTrack v2.0 es mcf06 (0.8%).

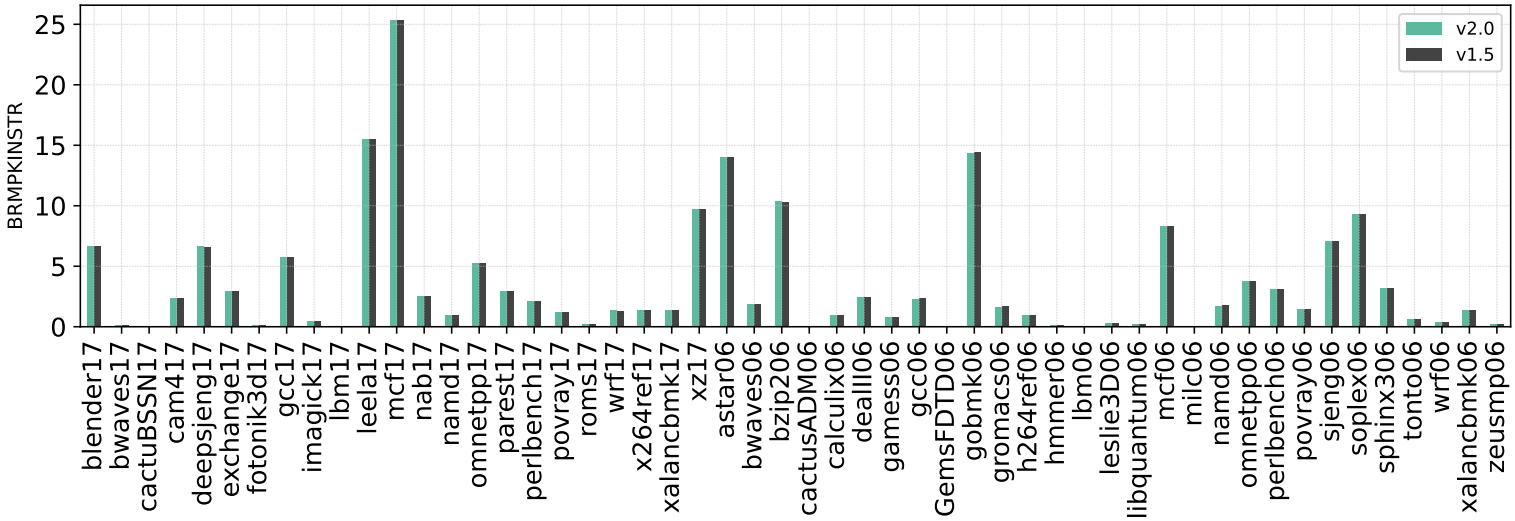
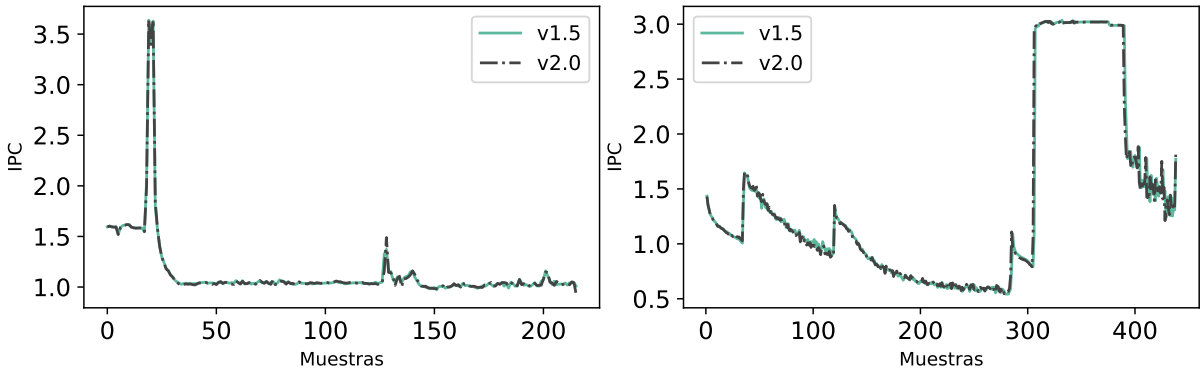


Figura 6.7: Media aritmética del BRMPKINSTR en modo TBS para los *benchmarks* de las suites SPEC CPU2006 y SPEC CPU2017

Con ambas gráficas (figura 6.6 y figura 6.7) se llegan a las mismas conclusiones que con las gráficas realizadas con los datos de monitorización obtenidos en modo EBS. La nueva implementación de la API de PMCTrack v2.0 no introduce sobrecarga en la precisión de las métricas en relación a la API presente en PMCTrack v1.5.

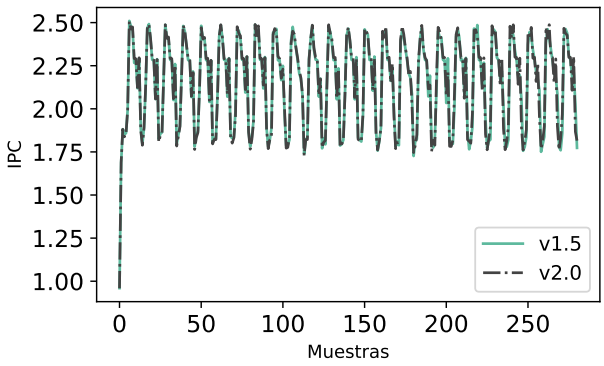
Para tener una visualización más detallada de las métricas evaluadas se ha escogido un conjunto específico de *benchmarks* de los SPEC CPU 2017 y 2006 con distintas áreas de aplicación. En la figura 6.8 se muestran las gráficas con el IPC de este conjunto de *benchmarks* y en la figura 6.9 se muestran las gráficas con su BRMPKINSTR.

En estas gráficas específicas de los *benchmarks* escogidos para el modo de monitorización TBS, se aprecia un ligero aumento de variabilidad para algunas muestras obtenidas. Al ser variaciones que se encuentran siempre en un rango inferior al 2% y analizadas en conjunto con la media de cada *benchmark* –representadas en las figuras 6.6 y 6.7–, se trata de una variabilidad que no es causada por PMCTrack. Todo esto demuestra que no se pierde precisión ni rendimiento en la monitorización con PMCTrack v2.0.

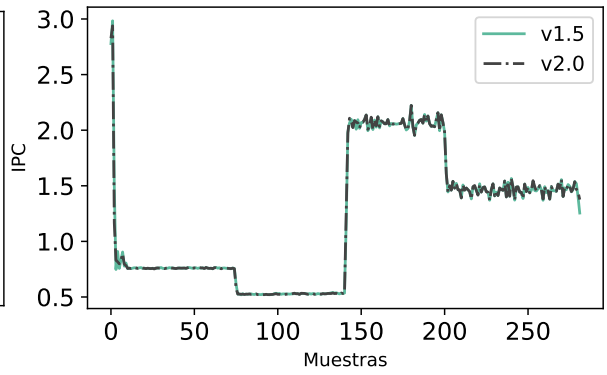


(a) *Benchmark* xz17

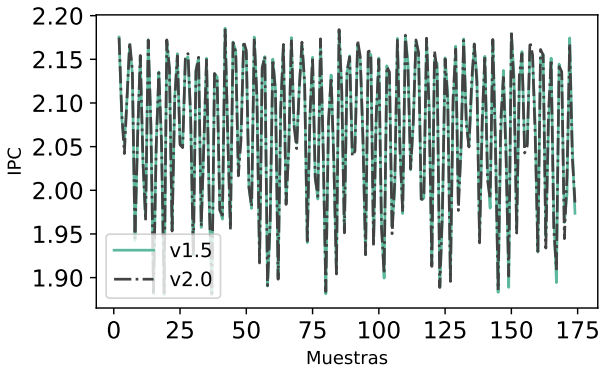
(b) *Benchmark* xalancbmk17



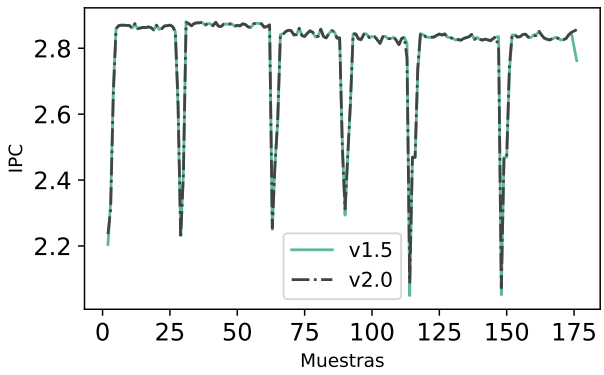
(c) *Benchmark* roms17



(d) *Benchmark* astar06

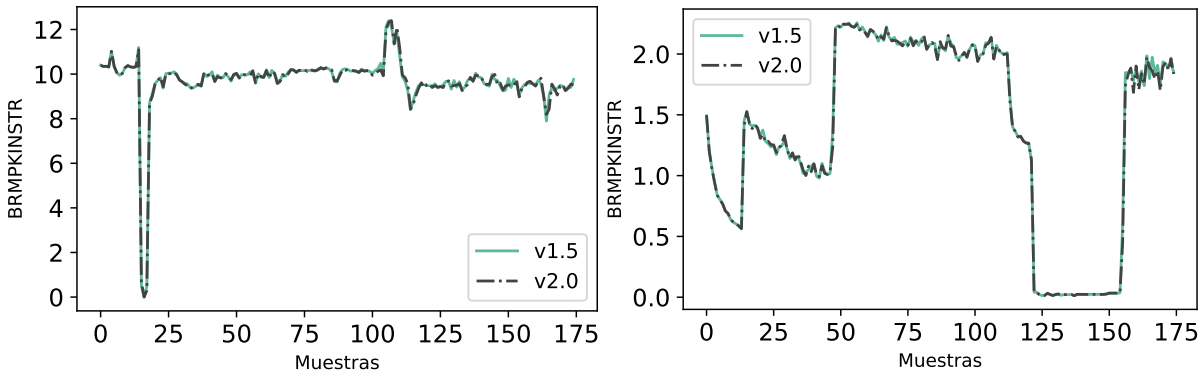


(e) *Benchmark* namd06



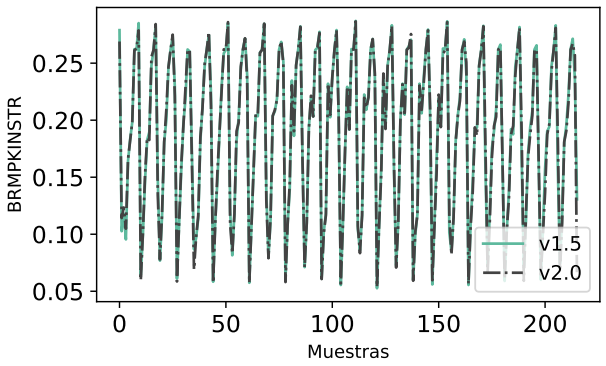
(f) *Benchmark* bwaves06

Figura 6.8: Evaluación del IPC en modo TBS

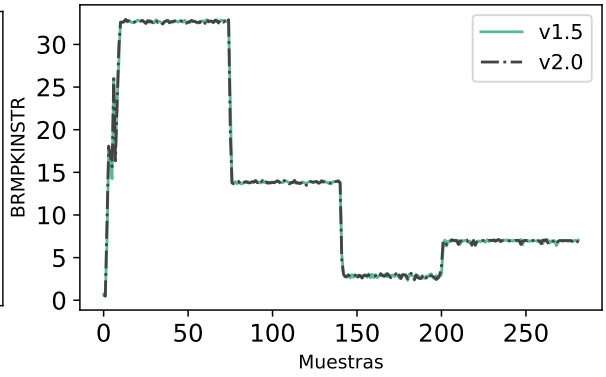


(a) *Benchmark* xz17

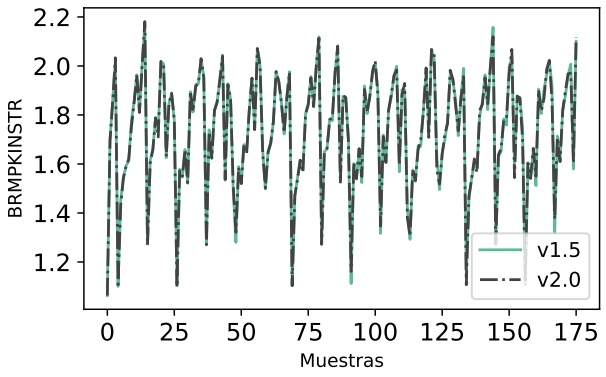
(b) *Benchmark* xalancbmk17



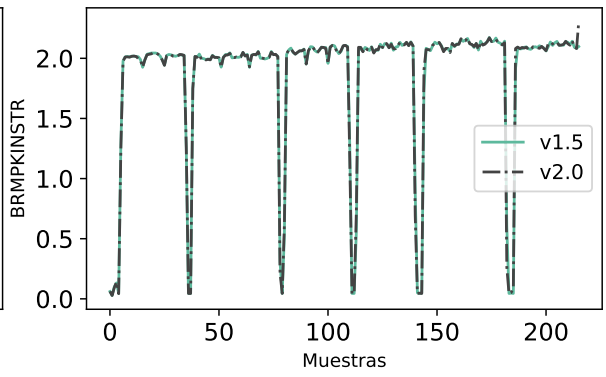
(c) *Benchmark* roms17



(d) *Benchmark* astar06



(e) *Benchmark* namd06



(f) *Benchmark* bwaves06

Figura 6.9: Evaluación del BRMPKINSTR en modo TBS

6.5 Evaluación de la sobrecarga en las callback de PMCTrack v2.0

La nueva manera de almacenar los datos de monitorización por hilo en PMCTrack v2.0 supone la modificación de la implementación de algunas de las callbacks dentro de la interfaz de callbacks de PMCTrack. Como se explica en la sección 4.3, estas modificaciones hacen uso de un evento *software* de perf_events para el mantenimiento de los datos de monitorización de PMCTrack. Esto supone que dentro de las callbacks ahora es necesario la creación, lectura y eliminación de dicho evento. El hecho de requerir el acceso a un evento de perf_events da lugar a pensar que se puede estar introduciendo una sobrecarga en la interfaz de callbacks de PMCTrack. Para evaluar si existe una sobrecarga se han medido los tiempos de ejecución de las tres callbacks más relevantes:

- La callback **alloc_per_thread_data**, en esta callback se produce la creación del nuevo evento *software* de perf_events.
- La callback **exit_thread**, se encarga de liberar el evento *software* de perf_events e insertar los datos de monitorización en una lista.
- La callback **free_per_thread_ data**, elimina los datos de monitorización de PMCTrack de la lista en que fueron insertadas por la callback **exit_thread**.

Estas callbacks, a pesar de no ser frecuentes, son las más relevantes porque son aquellas que han recibido las modificaciones más importantes con respecto a PMCTrack v1.5. Las llamadas más frecuentes dentro de PMCTrack son aquellas que se invocan cuando vence un temporizador en el modo TBS y cuando se produce la interrupción de desbordamiento en modo EBS. Estas llamadas no se invocan desde las callbacks de PMCTrack y no han sufrido modificaciones en la realización del proyecto. Es por eso que no han sido escogidas para la evaluación.

Para medir los tiempos de ejecución de cada callback (especificados en nanosegundos) se ha hecho uso de la herramienta bpftrace [90]. Esta herramienta, bpftrace, ofrece un lenguaje de trazado de alto nivel que ha permitido crear una serie de *scripts* capaces de hacer trazado sobre las callbacks de PMCTrack. Bpftrace utiliza la infraestructura del compilador LLVM [91] como base para compilar *scripts* a BPF-bytecode (*Berkeley Packet Filter*) y hace uso de BCC (BPF *Compiler Collection*) para interactuar con el sistema eBPF[4] [38] de Linux [90]. Bpftrace también hace uso de las capacidades de trazado existentes en Linux: trazado dinámico del kernel (kprobes), trazado dinámico a nivel de usuario (uprobes) y tracepoints [90].

En la tabla 6.1 se exponen la media, mínimo y máximo de los tiempos de ejecución de las callbacks obtenidos tanto para PMCTrack v2.0 y v1.5. Para obtener estos datos se han lanzado los *scripts* de bpftrace creados durante un intervalo de tiempo de 30 minutos mientras se realizaba la monitorización de los distintos *benchmarks* del SPEC

[4]eBPF es una tecnología revolucionaria que permite ejecutar programas en el kernel Linux sin cambiar el código fuente del kernel ni cargar módulos del kernel.

	v2.0			v1.5		
PMCTrack callback	Máx.	Mín.	Media	Máx.	Mín.	Media
alloc_per_thread_data	35346	3112	8476.116	29491	3504	8378.699
exit_thread	536311	4159	15248.827	610010	4074	15251.056
free_per_thread	28535	895	7012.519	26325	985	6979.068

Tabla 6.1: Tiempos de ejecución de la interfaz de callbacks de PMCTrack.

CPU2017 con la herramienta de línea de comandos `pmctrack`. El resultado ha sido una colección de más de 1400 muestras para cada callbacks.

Estos tiempos exponen una sobrecarga mínima en las callbacks **alloc_per_thread_data** y **free_per_thread**. Esta sobrecarga corresponde a menos de 0.2 microsegundos para la callback **alloc_per_thread_data** y de 0.03 microsegundos para la callback **free_per_thread**. Ambas sobrecargas se atribuyen al nuevo uso de operaciones de la API de perf_events dentro de estas callbacks y constituyen una diferencia insignificante de tiempo que no se puede considerar importante. Estas sobrecargas no se pueden considerar importantes porque ambas callbacks son invocadas una sola vez durante el ciclo de vida de un proceso, cuando se realiza su *fork* (**alloc_per_thread_data**) y cuando se libera su `task_struct` (**free_per_thread**).

En cuanto a la callback **exit_thread**, no sufre de sobrecarga con respecto a su implementación presente en PMCTrack v1.5. Esta callback presenta un tiempo medio ligeramente mejor, 0.5%, en la versión de PMCTrack v2.0. También ha reportado un máximo menor (13.7%) y mínimo mayor (2%) en comparación con PMCTrack v1.5.

Capítulo 7

Conclusiones y trabajo futuro

En este último capítulo se recogen las conclusiones finales de este proyecto y se enumeran los conocimientos adquiridos durante su realización. También se incluyen una serie de propuestas para proseguir el desarrollo de PMCTrack.

7.1 Resultados y conclusiones finales

El resultado de este proyecto es la próxima versión pública de PMCTrack v2.0[1]. Esta nueva versión incluye la nueva implementación de la API de PMCTrack a través de mecanismos de trazado que hace posible el uso de PMCTrack sobre Linux *vanilla*, sin la necesidad de inclusión de un parche específico de PMCTrack en el kernel. PMCTrack usará automáticamente la nueva versión de la API si detecta que no se encuentra instalado el parche de PMCTrack en el kernel del sistema, de lo contrario usará la API ya presente en la versión v1.5 y anteriores.

En PMCTrack v2.0, también se ha introducido la extensión realizada en el backend de perf_events de PMCTrack (módulo del kernel `mchw_perf`) dando nuevo soporte a arquitecturas ARM y AMD. Con esta nueva versión del backend de perf_events de PMCTrack también se ha ampliado considerablemente la cantidad de eventos de perf_events disponibles a monitorizar con el módulo del kernel de PMCTrack `mchw_perf`.

Todas estas nuevas inclusiones y mejoras introducidas en PMCTrack hacen que la nueva versión v2.0 eleven PMCTrack a un estado compatible con entornos de producción, cumpliendo así el ambicioso objetivo de este proyecto.

El nuevo soporte sin parche del kernel de PMCTrack presente en PMCTrack v2.0 se caracteriza por ser robusto y por no afectar al rendimiento ni la precisión de PMCTrack. Este soporte se ha usado durante incontables horas, además de puesto a prueba durante

[1]La versión PMCTrack v2.0 será publicada en el repositorio oficial de PMCTrack: https://github.com/jcsaezal/pmctrack

14 horas consecutivas sin reportar ninguna anomalía en su funcionamiento. Esto se debe en parte a la elección de sistemas de trazado con una gran madurez y diseñados para introducir la menor sobrecarga posible en el sistema. Lo expuesto muestra también como resultado una evaluación positiva sobre la nueva versión de la API de PMCTrack, con una sobrecarga prácticamente inexistente y completamente despreciable frente a la versión de la API de PMCTrack implementada mediante un parche del kernel (véase capítulo 6).

Para poder hacer uso de la nueva versión de la API de PMCTrack se precisa de tres requisitos en el sistema. El primero es la necesidad de que se encuentren activos los sistemas de trazado tracepoints, ftrace y perf_events dentro del sistema. Otro requisito se corresponde al uso de PMCTrack para ofrecer datos de monitorización a los componentes del kernel, como ya ha sido explicado en la sección 4.2.1, esta funcionalidad requiere usar la versión 5.9.7 de Linux o superiores. Cabe destacar que el uso de PMCTrack para la monitorización desde el espacio de usuario es posible utilizarlo sobre cualquier versión de Linux con soporte a los sistemas de trazado. El último requisito, se presenta sólo en arquitecturas ARM64 en las cuales el soporte de ftrace ha sido introducido recientemente y esto hace que el uso de la nueva versión de la API de PMCTrack esté sólo disponible a partir de Linux versión 5.5.19 para ARM64 (véase sección 5.4).

Durante la realización de este trabajo de fin de grado se han ido adquiriendo múltiples conocimientos nuevos y nuevas destrezas. En la creación de la nueva implementación de la API de PMCTrack, se ha logrado adquirir un conocimiento extenso de los sistemas de trazado presentes en Linux. Estos conocimientos corresponden a las prestaciones que ofrecen cada sistema de trazado, la funcionalidad que ofrecen, el alcance que tienen y su implementación. Este estudio ha hecho que se conozca en detalle a muy bajo nivel la implementación particular de los sistemas ftrace, tracepoints y kprobes.

En la búsqueda de conocimientos sobre la implementación de los sistemas mencionados anteriormente y durante la implementación de la API de PMCTrack se ha obtenido destreza en la búsqueda, interpretación y análisis del código fuente de Linux. (Cabe destacar lo extensas que son las fuentes del kernel Linux y su escasa documentación dentro de las fuentes –y fuera– por no decir su inexistencia en muchos casos.)

El estudio del uso de estos sistemas de trazado ha dado como resultado además una ampliación importante de conocimientos sobre la estructura y funcionamiento de Linux (planificación, expropiación, etc), al haber tenido que tratar con funciones del kernel que forman parte del núcleo del planificador de Linux.

Durante la implementación de la expansión del backend de perf_events de PMCTrack se ha adquirido conocimientos, externos a PMCTrack, sobre el subsistema de eventos perf_events, la herramienta `perf` y el subsistema de archivos sysfs en Linux.

Tanto el estudio realizado como la implementación de la nueva versión de la API y la extensión del backend de perf_events de PMCTrack, han llevado al conocimiento en profundidad y en detalle de PMCTrack.

Como aportación extra de este proyecto, se puede considerar la contribución de esta memoria como documentación exhaustiva en castellano de los sistemas de trazado junto con la recompilación de sus ventajas e inconvenientes.

7.2 Trabajo futuro

A continuación se enumeran una serie de posibles propuestas para continuar con el desarrollo de PMCTrack:

- **Continuación de la expansión del backend de perf_events de PMCTrack.**

  Con la ampliación de la funcionalidad del backend de perf_events realizada en este trabajo de fin de grado se ha introducido nuevo soporte para arquitecturas ARM y AMD. La posibilidad de incluir soporte a otras arquitecturas emergentes como RISC-V o ya establecidas como PowerPC ampliaría la gama de procesadores disponibles a monitorizar. Para lograr esto sería necesario la compatibilidad del subsistema de eventos perf_events para dichas arquitecturas. Con el rumor de que Intel planea lanzar procesadores asimétricos, Alder Lake S, en 2022 [92] –como los procesadores ARM big.LITTLE a los que se ha añadido soporte en este proyecto– también se puede plantear en un futuro añadir soporte para estos nuevos procesadores asimétricos.

- **Creación de un "puente" (*bridge*) entre perf_events y PMCTrack.**

  En el capítulo 5 se ha expuesto la limitación de perf_events de no tener mecanismos de descubrimiento de PMUs en su API del kernel. Esto implica que la configuración con la que se ha de crear un evento no es expuesta por perf_events dentro del espacio del kernel. Por el contrario, parte de la información de configuración de eventos de perf_events en formato *raw*, sí es expuesta a través del sysfs por el subsistemas de eventos perf_events o a través de la herramienta de línea de comandos `perf`. Esto ha resultado en una expansión del backend de perf_events para que soporte el uso de esta configuración en formato *raw* expuesta por perf_events en el espacio de usuario. Esto plantea el desarrollo de un "puente" entre perf_events y PMCTrack para que los mnemotécnicos que se usan para la herramienta `perf` estén disponibles también en PMCTrack.

  Una posible aproximación a la implementación de este *bridge* puede ser la creación de nuevos mnemotécnicos en las fuentes de PMCTrack en base a la información expuesta en el sysfs y a la que muestra la herramienta `perf` para los eventos no genéricos que se corresponden a ficheros *JSON* presentes en las fuentes de la herramienta `perf`.

  Otra aproximación que se puede plantear es el uso de la biblioteca libpfm4 [93]. Esta biblioteca esta diseñada para ayudar a convertir un nombre de evento, expresado como una cadena, en la codificación correspondiente al evento. Esta codificación

puede ser la configuración en formato *raw* del evento –tal como documenta el fabricante del procesador– o la codificación específica del sistema operativo. En este último caso, la biblioteca es capaz de preparar las estructuras de datos específicas del sistema operativo que necesita el kernel para configurar el evento [93], [94].

Apéndice A

Introduction

This introduction presents the motivation that has led to the approach of this project, it states the objectives to be achieved in the project and its planning. It also describes the structure of this document.

A.1 Motivation

Performance Monitoring Counters (PMCs) are a set of processor registers that allow the capture of events for the measurement of relevant performance metrics, such as instructions per cycle, miss rate of different cache levels or processor pipeline stall cycles caused by different reasons. A notable aspect of hardware counters is the fact that in today's processors, they are only directly accessible from the privilege level where the operating system (OS) typically runs. This makes it necessary to have specific support in the operating system, implemented as part of the kernel or in a dedicated driver, to access monitoring information from user space.

Among the various tools that allow access to PMCs one of them is PMCTrack [1]. PMCTrack is an open source tool for GNU/Linux whose development began in 2007. PMCTrack is the only monitoring tool designed from the ground up to provide data from PMCs, as well as other hardware monitoring sources of a system, to OS components. This monitoring information is exposed in a straightforward manner via a kernel API that leverages architecture-independent abstractions. Also, like other hardware counter management tools, such as `perf` or Intel Performance Counter Monitor [2], [3], PMCTrack also allows the collection of monitoring data from user space. For this purpose it provides the `pmctrack` program –a command line tool–, PMCTrack-GUI –a graphical frontend– and libpmctrack –a library that allows instrumentation of C/C++ code–. It is also worth noting that any kind of relevant monitoring information provided by the hardware, but not directly exposed through PMCs (such as power consumption, cache occupancy, or number of page faults, etc.), can also be exposed to the OS and end-user through

PMCTrack's virtual counter abstraction [1].

This project focuses on the extension of the PMCTrack functionality implemented at the kernel level. For the correct performance monitoring of each individual application using PMCs, the OS kernel has to manage the registers associated with these hardware monitoring resources. More specifically, when a thread whose performance is being monitored exits the CPU (e.g., is blocked by input-output or synchronization) the context of the hardware counters must be saved. Similarly, this context has to be restored to the appropriate CPU when the OS gives the thread the opportunity to execute instructions again. This shows that the hardware counter management code is closely linked to that of the Linux kernel component that performs context switches: the process scheduler.

With that in mind, in order to maintain PMCs and other hardware monitoring logs directly, any monitoring tool must be fully aware of critical scheduling events (such as process creation, process termination, occurrence of context switches, etc.). This implies that, for the maintenance of PMCs in PMCTrack, specific changes in the Linux scheduler are required to make it possible to "capture" these scheduler events. Although making changes to the kernel is the most natural way to achieve direct access to the hardware counters, it also brings a major limitation: any future changes to the counter management tool require kernel recompilation, and a system reboot for testing. This can make the development process extremely slow.

Unlike a hardware counter management tool that is implemented directly in the Linux kernel, as is the case with the `perf` tool that has been introduced in the Linux mainline –with its consequent high maintenance cost– PMCTrack divides its kernel-level functionality into two parts:

- **An API (Application Programming Interface) in the kernel.** This API consists of (1) a series of calls in the scheduler to capture scheduling events–it forms a callback interface–, (2) a specific API to associate callbacks to these events from other kernel components, (3) functions to access per-thread PMCs data from within the kernel and (4) the inclusion of extra fields in the process descriptor (`struct task_struct`) for maintaining monitoring information. This PMCTrack API is introduced in the kernel through a patch. Therefore, to use PMCTrack on a system it is necessary to have a modified kernel with this patch.

  It should be noted that the PMCTrack kernel patch does not involve the inclusion of major changes to the Linux sources; only two new source files have to be incorporated and a few dozen lines of code have to be added to the Linux scheduler sources and related files [1].

- **A kernel module that implements most of the functionality of PMCTrack.** This kernel module, when loaded, installs a callback interface to receive the necessary scheduler notifications that enables the management of the hardware monitoring registers. It is noteworthy that access to performance hardware counters on different

architectures and processor models in PMCTrack is abstracted to other PMCTrack software layers, through the use of backends –one for each architecture–. At compile time, different variants of the kernel module are generated for each architecture [4].

The fact that most of PMCTrack's functionality is included in a kernel module greatly simplifies the maintenance and debugging of this tool. Any new functionality can be included in the kernel module code, and quickly tested by loading and unloading the module into Linux as many times as necessary, without having to reboot the system[1]. Despite its ease of maintenance, PMCTrack has two relevant limitations, which hinder its widespread adoption in production environments:

1. **Dependence of a Linux kernel patch to work.**

   This involves the creation and release, by PMCTrack developers, of a Linux kernel patch specific to each Linux version and architecture on which you want to work. A user intending to use PMCTrack on a Linux architecture and version for which an official PMCTrack patch already exists, must have basic knowledge in Linux kernel compilation to achieve the correct installation of the patch and kernel compilation.

   In the absence of a PMCTrack patch released by the PMCTrack developers, a user who wants to use PMCTrack must have, in addition to knowledge of kernel compilation, advanced knowledge of PMCTrack and the Linux kernel. After all, the correct implementation of a patch –based on an existing patch for a previous Linux version– requires tracking changes made to the scheduler code in the new version to determine where to include the new lines of code in the patch (mainly notifications to the kernel module of the occurrence of scheduling events).

2. **Partial integration with Linux perf_events subsystem.**

   Hardware counter management tools created prior to the 2010s, such as PMCTrack, are characterized for performing a direct access to the hardware monitoring registers [5]. As will be discussed in detail in chapter 2, this has historically caused incompatibility between different counter management tools on Linux. To address this problem, the Linux perf_events subsystem [6] was created, which provides a low-level API to perform counter access and management by abstracting most architecture-specific aspects.

   A first step for the integration between PMCTrack and perf_events, was the creation of a generic backend (*mchw_perf*) in which the access to PMCs is not done directly by reading the PMU (Performance Monitoring Unit) registers –as in the rest of PMCTrack backends–, instead it is done using the perf_events API [7]. However, this new backend, incorporated in the latest stable version of PMCTrack (v1.5) is not complete and lacks many basic functionalities of PMCTrack. In particular, it has only support for some Intel x86 processors, implements only

[1] Only critical bugs in the implementation that cause a kernel panic force such reboots

PMCTrack's user-space functionality, and supports only a very small set of the events offered by the Linux perf_events subsystem.

These two limitations make the widespread adoption of PMCTrack in production systems difficult, and the creation of support for new architectures and processor models can become a slow and costly process. In production systems, stable kernels with security updates are essential to provide the necessary robustness demanded by such systems. Requiring a modified kernel instead of a stable (longterm) kernel greatly complicates the application of security updates, which leads to justified reticence on the behalf of system administrators. On the other hand, to incorporate support for new architectures and processor models in PMCTrack, it was traditionally necessary to create a new backend or modify an existing one, to perform low-level access to the hardware counters using assembly code (embedded in "C" code). The creation of the new perf_events backend in PMCTrack v1.5 –although it is limited– represents a great opportunity to offer PMCTrack support more quickly to new architectures; because perf_events is maintained by employees of major hardware manufacturers to support a wide spectrum of architectures and processor models.

PMCTrack is the only tool that currently offers easy architecture-independent access to hardware monitoring data within Linux kernel components. This means that there is a huge motivation to adopt it to production systems, as well as to simplify the process of including support for new architectures and processor models. While this can be a major challenge–and the elimination of historical limitations of PMCTrack–there are two factors that make it feasible to achieve at the present time. First of all, the Linux kernel currently implements a set of tracing technologies –such as tracepoints [8], Kprobes [9] and ftrace [10]– that have reached a high level of maturity and are supported on a large number of architectures. This type of technologies allow, among other things, to know what is going on inside Linux (which functions are executed and in which order) or even to modify the kernel behavior at runtime (e.g. livepatching [11]) by installing callbacks at certain points in the kernel code. The use of these tracing technologies raises the possibility of eliminating the limitation of having to apply a PMCTrack patch to the kernel. Moreover, the perf_events backend in PMCTrack v1.5, despite its limitations, provided an excellent proof of concept that using this kernel subsystem to prevent low-level access to PMCs is possible [7]. Therefore, extending the functionality of this backend is a great incentive to accelerate the process of adding support for other processor architectures and models–particularly ARM and AMD–as well as to allow access from PMCTrack to a wider range of the monitoring events offered by perf_events.

A.2 Objectives

The main objective of this project is to adapt the PMCTrack tool to production systems. Achieving this involves achieving the following three sub-objectives:

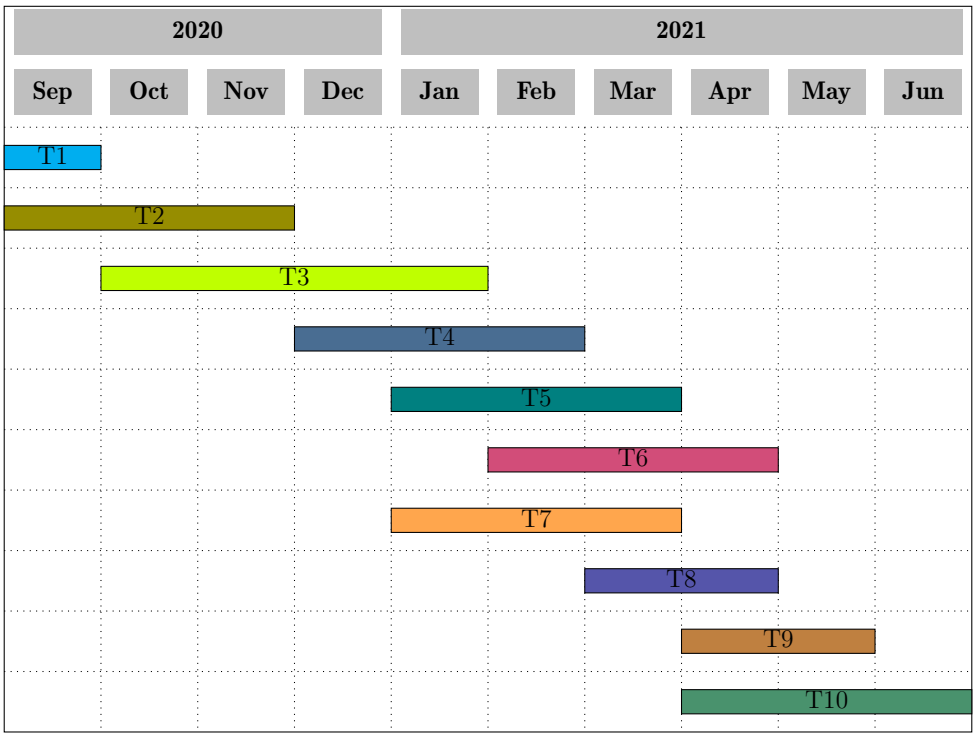


Figura A.1: Time spent for each task performed during the project

1. The adaptation of the PMCTrack API installation in the Linux kernel to an implementation that does not require modifying the kernel source code. This requires the exhaustive study, the choice and the combined application of the different existing tracing technologies in the mainline Linux, as well as the analysis of the different alternatives to achieve the most efficient implementation.
2. Increase the architectures supported by the experimental perf_events backend included in PMCTrack v1.5. This means the extension of this backend to new architectures, such as ARMv7 and ARMv8, as well as the inclusion of support for new AMD processors.
3. Expand the functionality of the experimental perf_events backend. This implies a substantial increase in the number of events of the perf_events subsystem that could be monitored with PMCTrack, both from user space and kernel components.

A.3 Work schedule

In the scheduling of this project, a series of tasks have been established to achieve the stated objectives. The time taken to complete each task is represented in the Gantt chart in Figure A.1. Each task in the diagram is described below and is identified by a number in the following list:

T1. Project approach. This task corresponds to the pre-development phase. It identified

the limitations of PMCTrack and their need to be removed in order to convert PMCTrack into a version suitable for production systems. It was studied how to divide the PMCTrack kernel patch into parts in order to be able to deal in order with the parts of the patch that require different adaptations. A pool of possible technologies to be used in the project was made and the order in which the parts that the PMCTrack patch was divided into was planned.

T2. Study of tracing technologies. In this task, the study of the tracing systems (tracepoints [8], Kprobes [9] and ftrace [10]) was conducted. It was determined whether they offer the functionality and efficiency needed to address the porting of the kernel PMCTrack API to an implementation that does not require a kernel patch. This phase includes the creation of kernel modules to test the behavior of such tracing systems, so that it could be evaluated which technologies are the most suitable.

T3. Implementation of the PMCTrack API callback interface adaptation. This task entailed the application of those chosen tracing systems to adapt the PMCTrack API callback interface to an implementation without the need for a Linux patch. The implementation was made with the idea that it would work with both a vanilla Linux and a modified kernel with the PMCTrack API. Thus maintaining backwards compatibility and compatibility with other PMCTrack backends.

T4. Study of the Linux perf_events subsystem. At this stage, the possibilities offered by the perf_events subsystem to maintain the necessary monitoring data for PMCTrack in the information associated with each Linux task were studied. This study continued in parallel for the extension of the experimental backend of perf_events.

T5. Storage of monitoring data per thread in vanilla kernels. In this task the implementation, using perf_events, of the new way in which PMCTrack stores the monitoring data per task, so that it does not require a Linux kernel patch, was done. As with the PMCTrack callback interface, this implementation was also done with the idea of maintaining backward compatibility.

T6. Incorporation of backward compatibility in the architecture-independent layer. This task consisted of performing the last steps of the adaptation to the new implementation of the PMCTrack API, so that its architecture-independent layer would continue to work correctly regardless of the kernel used. Compatibility had to be maintained whether PMCTrack runs on the new PMCTrack API implementation or on kernels patched with the old implementation.

T7. Extension of the perf_events backend functionality. At this phase, the new implementation of the PMCTrack API and the experimental backend of perf_events were adapted to create support for ARM and AMD processors. This also involved the extension of the experimental perf_events backend to achieve a much wider support of the events available in perf_events.

T8. Compatibility of the new implementation with legacy backends. In this task the necessary modifications were implemented to achieve compatibility with old PMCTrack backends (`intel-core`, `amd`, `arm`, `odroid-xu`, etc.). This resulted in the correct functioning of all PMCTrack backends published with the new PMCTrack API implementation.

T9. Experimental evaluation and analysis. A study was made about the possible overhead incorporated by the use of tracing systems for the new implementation of the PMCTrack API. Experimental analysis of this overhead was performed using benchmarks of the SPEC CPU2006 and CPU2017 suites.

T10. Preparation of the project documentation. In this period, this project documentation was written and reviewed on an on-going basis.

A.4 Document organization

The following is a brief description of the different chapters that make up the rest of this document:

- **Chapter 2** describes the historical context and architecture of PMCTrack. It also explains in detail the PMCTrack patch, the PMCTrack kernel API, and its division into parts to address the project.
- **Chapter 3** presents the historical context of tracing systems in Linux and describes the types of tracing that exist within the Linux kernel. It also exposes the existence of trace systems within the main branch of the Linux kernel that allow tracing over the running Linux source code. It also includes the explanation of the inner workings of these tracing systems and the argumentation of the functionalities they offer in relation to their application in the adaptation of the PMCTrack API.
- **Chapter 4** explains in detail the implementation of the new version of the PMCTrack API compatible with production systems. The implementation is done by using the studied tracing systems, ftrace, tracepoints and perf_events.
- **Chapter 5** provides a detailed description of the changes made to the experimental perf_events backend to extend its functionality and support for processors beyond Intel x86. It also presents the changes made to the new implementation of the PMCTrack API to provide support for other architectures (AMD and ARM) and for the other backends available in PMCTrack.
- **Chapter 6** covers the experimental results obtained using benchmarks of the SPEC CPU2006 and CPU2007 suites. A detailed analysis of the accuracy of the performance metrics obtained and the overhead associated with the new implementation of the PMCTrack API callback interface is also carried out.
- **Chapter 7** highlights the advantages of the new version of PMCTrack. The

conclusions and knowledge acquired during the development of this project are also presented. Different ideas for the possible continuation of the project are included.

- The English translation of the introduction (Chapter 1) and the conclusions (Chapter 7) can be found in **Appendices A and B**, respectively.

Apéndice B

Conclusions and future work

This last chapter contains the final conclusions of this project and lists the knowledge acquired during its implementation. It also includes a series of proposals for the further development of PMCTrack.

B.1 Final results and conclusions

The result of this project is the next public version of PMCTrack, version v2.0[1]. This new version includes the new implementation of the PMCTrack API through tracing mechanisms that makes it possible to use PMCTrack on vanilla Linux, without the need of including a PMCTrack specific patch in the kernel. PMCTrack will automatically use the new API version if it detects that the PMCTrack patch is not installed in the system's kernel, otherwise it will use the API already present in version v1.5 and earlier.

In PMCTrack v2.0, the extension made in the PMCTrack perf_events backend (kernel module `mchw_perf`) has also been introduced, giving new support to ARM and AMD architectures. With this new version of the PMCTrack perf_events backend, the number of perf_events events available to monitor with the PMCTrack kernel module `mchw_perf` has also been considerably extended.

All these new inclusions and improvements introduced in PMCTrack make the new version v2.0 bring PMCTrack to a state compatible with production systems, thus fulfilling the ambitious goal of this project.

The new PMCTrack kernel patchless support in PMCTrack v2.0 is robust and does not affect PMCTrack's performance and accuracy. This support has been used for countless hours and has been tested for 14 consecutive hours without reporting any performance anomalies. This is due in part to the choice of highly mature tracing systems designed

[1]PMCTrack v2.0 will be published in the official PMCTrack repository: https://github.com/jcsaezal/pmctrack

to introduce as little overhead as possible into the system. The above also results in a positive evaluation of the new version of the PMCTrack API, with virtually no overhead and completely negligible overhead compared to the version of the PMCTrack API implemented through a kernel patch (chapter 6)

In order to make use of the new version of the PMCTrack API, three system requirements are necessary. The first one is the need for the tracepoints, ftrace and perf_events tracing systems to be active within the system. Another requirement corresponds to the use of PMCTrack to provide monitoring data to kernel components, as already explained in section 4.2.1, this functionality requires using Linux version 5.9.7 or higher. It should be noted that the use of PMCTrack for userspace monitoring is possible on any version of Linux with support for tracing systems. The last requirement is present only on ARM64 architectures on which ftrace support has been recently introduced, this makes the use of the new PMCTrack API only available for ARM64 from Linux version 5.5.19 onwards (section 5.4).

Throughout the course of this project, a significant amount of new knowledge and new skills have been acquired. During the creation of the new implementation of the PMCTrack API, it has been obtained an extensive knowledge of the tracing systems present in Linux. This knowledge corresponds to the features offered by each tracing system, the functionality they offer, the scope they have and their implementation. This study has made possible the acquisition of detailed knowledge at a very low level about the particular implementation of ftrace, tracepoints and kprobes systems.

In the pursuit of knowledge about the implementation of the systems mentioned above and during the implementation of the PMCTrack API, skills have been gained in searching, interpreting and analyzing Linux source code. (It is worth noting how extensive the Linux kernel sources are and how little documentation there is within the sources –and outside– if any at all in many cases).

The study of the use of these tracing systems has also resulted in a significant expansion of knowledge about the structure and operation of Linux (scheduling, preemption, etc), due to having to deal with kernel functions that are part of the Linux scheduler core

During the implementation of the PMCTrack perf_events backend expansion, knowledge has been acquired, external to PMCTrack, about the perf_events event subsystem, the `perf` tool and the sysfs file subsystem in Linux.

Both the study carried out and the implementation of the new version of the API and the extension of the PMCTrack perf_events backend, have led to an in-depth and detailed knowledge of PMCTrack.

As an extra result, the contribution of this project's document can also be considered as an exhaustive documentation in Spanish of the tracing systems along with the recompilation of their advantages and disadvantages.

B.2 Future work

Listed below are a number of possible proposals for further development of PMCTrack:

- **Continuation of the expansion of the perf_events backend of PMCTrack.**

  With the expansion of the perf_events backend functionality in this project, new support for ARM and AMD architectures has been introduced. TThe possibility of including support for other emerging architectures such as RISC-V or established architectures such as PowerPC would expand the range of processors available for monitoring. To achieve this, perf_events subsystem support for these architectures would be required. With Intel rumored to be planning to release asymmetric processors, Alder Lake S, in 2022 [92] –like the ARM big.LITTLE processors for which support has been added in this project– adding support for these new asymmetric processors may also be considered in the future.

- **Creation of a "bridge" between perf_events and PMCTrack.**

  In chapter 5 the limitation of perf_events of not having PMU discovery mechanisms in its kernel API has been exposed. This implies that the configuration with which a specific event is to be created is not exposed by perf_events within the kernel space. In contrast, some of the perf_events event configuration information in raw format is exposed through the sysfs by the perf_events event subsystem or through the `perf` command line tool. This has resulted in an expansion of the perf_events backend to support the use of this raw format configuration exposed by perf_events in user space. This raises the development of a "bridge" between perf_events and PMCTrack so that the mnemonics used for the `perf` tool are also available in PMCTrack.

  A possible approach to the implementation of this bridge could be the creation of new mnemonics in the PMCTrack sources based on the information displayed in the sysfs and provided by the `perf` tool for non-generic events, that corresponds to *JSON* files present in the `perf` tool sources.

  Another approach that can be considered is the use of the libpfm4 library [93]. This library is designed to help convert an event name, expressed as a string, into the encoding corresponding to the event. This encoding can be the raw format configuration of the event–as documented by the processor manufacturer–or the operating system specific encoding. In the latter case, the library is able to prepare the operating system-specific data structures needed by the kernel to configure the event [93], [94].

Bibliografía

[1] J. C. Saez, A. Pousa, R. Rodriíguez-Rodriíguez, F. Castro, y M. Prieto-Matias, «PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler», *The Computer Journal*, vol. 60, n.º 1, pp. 60-85, ene. 2017.

[2] «perf: Linux profiling with performance counters». https://perf.wiki.kernel.org/index.php/Main_Page, 12-jun-2020.

[3] Thomas WillhalmR. D., «Intel® Performance Counter Monitor - A Better Way to Measure CPU Utilization». https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html, 05-ene-2017.

[4] «PMCTrack github». https://github.com/jcsaezal/pmctrack.

[5] V. M. Weaver, «Linux perf event Features and Overhead», *FastPath Workshop*, abr. 2013.

[6] V. M. Weaver, «The Unofficial Linux Perf Events Web-Page».

[7] Sáez de Buruaga BrounsJ., «Desarrollo de extensiones para herramientas de monitorización de rendimiento de código abierto», 2020.

[8] «Using the Linux Kernel Tracepoints». https://www.kernel.org/doc/html/latest/trace/tracepoints.html.

[9] «Kernel Probes (Kprobes)». https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[10] «ftrace - Function Tracer». https://www.kernel.org/doc/html/latest/trace/ftrace.html.

[11] «Livepatch». https://www.kernel.org/doc/html/latest/livepatch/livepatch.html, 25-abr-2021.

[12] «Pentium wiki». https://en.wikipedia.org/wiki/Pentium#Pentium.

[13] J. Levon, «Oprofile». http://oprofile.sourceforge.net.

[14] S. Jarp, R. Jurga, y A. Nowak, «Perfmon2: a leap forward in performance monitoring», *Journal of Physics: Conference Series*, vol. 119, p. 042017, jul. 2008.

[15] Sáez AlcaideJ. C., «Planificación de procesos en sistemas multicore asimétricos = Thread Scheduling on Asymmetric Multicore Systems», Tesis doctoral, Universidad Complutense de Madrid, Servicio de Publicaciones, Madrid, 2011.

[16] S. Eranian, «2.6.16 perfmon2 new code base + libpfm available». https://lwn.net/Articles/176647/, 22-mar-2006.

[17] T. Hoare, «Oprofile - Additional patches». https://oprofile.sourceforge.io/patches/, 20-jul-2020.

[18] T. Gleixner, «[Announcement] Performance Counters for Linux». https://lwn.net/Articles/310176/, 04-dic-2008.

[19] J. Edge, «Perfcounters added to the mainline». https://lwn.net/Articles/339361/, 01-jul-2009.

[20] A. C. de Melo, «The New Linux 'perf' tools». http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf, sep-2010.

[21] S. Eranian, «Re: [patch 0/3] [Announcement] Performance Counters for Linux». https://lwn.net/Articles/310176/, 06-dic-2008.

[22] P. Mackerras, «Re: [patch 0/3] [Announcement] Performance Counters for Linux». https://lwn.net/Articles/310273/, 06-dic-2008.

[23] Martínez FernándezG., Sánchez GordoS., y Dronda MerinoS., «Interfaz de uso de contadores hardware multiplataforma», 2012.

[24] J. Corbet, «Raw events and the perf ABI». https://lwn.net/Articles/441209/, 03-may-2011.

[25] J. C. Saez *et al.*, «An OS-Oriented Performance Monitoring Tool for Multicore Systems», en *Euro-Par 2015: Parallel Processing Workshops*, 2015, pp. 697-709.

[26] «What is RCU? – "Read, Copy, Update"». https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html.

[27] «Locking lessons». https://www.kernel.org/doc/html/latest/locking/spinlocks.html.

[28] I. Molnar, «[PATCH] LTT for 2.5.38 1/9: Core infrastructure». https://lore.kernel.org/lkml/3D8E1AF8.91C1915D@opersys.com/T/#m2cea7cab3f7e7b6720a00645981582e000130e77, 22-sep-2002.

[29] S. Rostedt, «Unified Tracing Platform - Bringing tracing together». https://static.sched.com/hosted_files/osseu19/5f/unified-tracing-platform-oss-eu-2019.pdf, 2019.

[30] R. Moore, «A Universal Dynamic Trace for Linux and other Operating Systems». https://www.usenix.org/legacy/event/usenix01/freenix01/full_papers/moore/moore.pdf, jun-2001.

[31] I. Molnar, «New release of LTT». https://www.opersys.com/LTT/news.html#18-11-1999, 18-nov-1999.

[32] Ananth Mavinakayanahalli Prasanna PanchamukhiJ. K., «Probing the Guts of Kprobes». https://www.kernel.org/doc/ols/2006/ols2006v2-pages-109-124.pdf, jul-2006.

[33] S. Rostedt, «Realtime Preemption». https://elinux.org/Realtime_Preemption, 21-dic-2004.

[34] «Kernel markers». https://lwn.net/Articles/245671/.

[35] «Kernel marker». https://en.wikipedia.org/wiki/Kernel_marker.

[36] J. Corbet, «KS2008: Tracing». https://lwn.net/Articles/298685/, 17-sep-2008.

[37] «Berkeley Packet Filter (BPF)». https://www.kernel.org/doc/html/latest/bpf/index.html.

[38] «eBPF Documentation». https://ebpf.io/.

[39] J. Evans, «Linux tracing systems & how they fit together». https://jvns.ca/blog/2017/07/05/linux-tracing-systems/#ftrace, jun-2017.

[40] V. Hsiao, «Linux kernel tracing». https://www.slideshare.net/vh21/linux-kernel-tracing, 09-feb-2016.

[41] «Notes on Analysing Behaviour Using Events and Tracepoints». https://www.kernel.org/doc/html/latest/trace/tracepoint-analysis.html.

[42] «Hooking into the kernel: real-time code execution at kernel level». https://hugoguiroux.blogspot.com/2016/01/hooking-into-kernel-real-time-code.html.

[43] «Hacking Linux USDT with Ftrace». http://www.brendangregg.com/blog/2015-07-03/hacking-linux-usdt-ftrace.html.

[44] «lttng-ust — LTTng user space tracing». https://lttng.org/man/3/lttng-ust/v2.10/.

[45] «Tracing the Linux kernel with ftrace». https://embeddedbits.org/tracing-the-linux-kernel-with-ftrace/.

[46] G. team, «GCC, the GNU Compiler Collection». https://gcc.gnu.org/.

[47] «ftrace». https://en.wikipedia.org/wiki/Ftrace.

[48] S. Rostedt, «Ftrace Kernel Hooks: More than just tracing». https://blog.linuxplumbersconf.org/2014/ocw/system/presentations/1773/original/ftrace-kernel-hooks-2014.pdf, 2014.

[49] S. Rostedt, «ftrace: Where modifying a running kernel all started». https://kernel-recipes.org/en/2019/talks/ftrace-where-modifying-a-running-kernel-all-started/, 2019.

[50] «Linux Static Tracepoints». https://sourceware.org/systemtap/kprobes/.

[51] «Linux Kernel Markers». https://lttng.org/docs/v2.12/.

[52] «The LTTng Documentation». https://kernelnewbies.org/Linux_2_6_24#Linux_Kernel_Markers.

[53] «Livepatch». https://www.kernel.org/doc/html/latest/livepatch/livepatch.html.

[54] A. Lozovsky, «Hooking Linux Kernel Functions, Part 3: What Are the Main Pros and Cons of Ftrace?» https://www.apriorit.com/dev-blog/547-hooking-linux-functions-3, 29-jun-2018.

[55] «blktrace(8) — Linux manual page». https://www.man7.org/linux/man-pages/man8/blktrace.8.html.

[56] A. Lozovsky, «Hooking Linux Kernel Functions, Part 1: Looking for the Perfect Solution». https://www.apriorit.com/dev-blog/544-hooking-linux-functions-1, 29-jun-2018.

[57] «SystemTap wiki». https://sourceware.org/systemtap/wiki, 08-may-2021.

[58] «Kernel Probes». https://anton.ozlabs.org/blog/2009/10/07/linux-static-tracepoints/.

[59] B. Gregg, «perf Examples». http://www.brendangregg.com/perf.html.

[60] «perf (Linux)». https://en.wikipedia.org/wiki/Perf_%28Linux%29, 23-dic-2020.

[61] B. Gregg, «Linux uprobe: User-Level Dynamic Tracing». http://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html, 28-jun-2015.

[62] «perf-list(1) — Linux manual page». https://www.man7.org/linux/man-pages/man1/perf-list.1.html.

[63] J. C. Saez, J. Gomez, y Prieto MatiasM., «Improving Priority Enforcement via Non-Work-Conserving Scheduling», 2008, pp. 99-106.

[64] «Taming Tracepoints in the Linux Kernel». https://blogs.oracle.com/linux/taming-tracepoints-in-the-linux-kernel.

[65] «clone(2) — Linux manual page». https://www.man7.org/linux/man-pages/man2/clone.2.html.

[66] «Using ftrace to hook to functions». https://www.kernel.org/doc/html/latest/trace/ftrace-uses.html.

[67] G. Kroah-Hartman, «Linux 5.9.7 release». https://lwn.net/Articles/836772/, 10-nov-2020.

[68] J. Corbet, «Four short stories about preempt_count()». https://lwn.net/Articles/831678/, 18-sep-2020.

[69] G. Kroah-Hartman, «[PATCH 5.4 37/85] ftrace: Handle tracing when switching between context». https://www.spinics.net/lists/stable/msg426317.html, 09-nov-2020.

[70] S. Rostedt, «ftrace: Have callbacks handle their own recursion». https://lwn.net/Articles/835848/, 30-oct-2020.

[71] A. F. T. U. of Denmark, «Calling conventionsfor different C++ compilers and operating systems». https://www.agner.org/optimize/calling_conventions.pdf, 2021.

[72] «RCU Concepts». https://www.kernel.org/doc/html/latest/RCU/rcu.html.

[73] V. M. Weaver, «perf_event_open(2) — Linux manual page». https://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[74] P. Zijlstra, «perf: sysfs type id». https://lwn.net/Articles/414490/, 09-nov-2010.

[75] «Processing Architecture for Power Efficiency and Performance». https://www.arm.com/why-arm/technologies/big-little.

[76] «AMD EPYC™ 7002 Series Processors». https://www.amd.com/en/processors/epyc-7002-series.

[77] «Opteron™ X and A-Series Processors». https://www.amd.com/en/opteron.

[78] «libpfm_perf_event_raw(3) — Linux manual page». https://man7.org/linux/man-pages/man3/libpfm_perf_event_raw.3.html.

[79] «ODROID-XU4 USER MANUAL». https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf.

[80] «HiKey960 Development Board User Manual». https://www.96boards.org/documentation/consumer/hikey/hikey960/hardware-docs/hardware-user-manual.md.html.

[81] J. Low, «[PATCH v3] MCS spinlock: Use smp_cond_load_acquire() in spin loop». https://lkml.org/lkml/2016/4/20/725, 20-abr-2016.

[82] T. Duwe, «[PATCH 1/3] arm64: implement ftrace with regs». http://lkml.iu.edu/hypermail/linux/kernel/1808.1/02248.html, 10-ago-2018.

[83] T. Duwe, «[PATCH v8] add -fpatchable-function-entry=N,M option». https://gcc.gnu.org/legacy-ml/gcc-patches/2017-05/msg00213.html, 03-may-2017.

[84] L. Bin, «livepatch: add support on arm64». https://lwn.net/Articles/646317/, 28-may-2015.

[85] T. Duwe, «[v4,1/3] arm64: implement ftrace with regs». https://patchwork.kernel.org/project/linux-arm-kernel/patch/20181026142148.6353A68C94@newverein.lst.de/, 26-ago-2018.

[86] S. Jennings, «Kernel Live Patching». https://lwn.net/Articles/619390/, 06-nov-2014.

[87] S. P. E. Corporation, «SPEC CPU® 2017». https://www.spec.org/cpu2017/.

[88] «SPEC CPU 2017 & Changing Performance». https://www.amd.com/system/files/2017-06/SPEC-CPU-2017-and-Changing-Performance.pdf, jun-2017.

[89] «Mispredicted Branches Retired». http://qcd.phys.cmu.edu/QCDcluster/intel/vtune/reference/Mispredicted_Branches_Retired.htm.

[90] «bpftrace - Github». https://github.com/iovisor/bpftrace.

[91] «The LLVM Compiler Infrastructure». https://llvm.org/.

[92] «Intel planning big.LITTLE for desktop?» https://videocardz.com/newz/intel-alder-lake-s-to-feature-16-cores-125-150w-tdp-and-pcie-4-0, 08-mar-2020.

[93] «libpfm-4.x - Github». https://github.com/wcohen/libpfm4.

[94] B. Nikolic, «How to monitor the full range of CPU performance events». http://www.bnikolic.co.uk/blog/hpc-prof-events.html.